Java:

初识命令行cmd:

常见的cmd的命令

1. 盘符名称+： 切换盘
2. dir 查看当前路径下的内容
3. cd 目录 切换到单级目录
4. Cd .. 回退到上一级的目录
5. cd 目录1\目录2\目录3\ 切换多级目录
6. Cd \回退到我们的盘符
7. Cls 清屏
8. exit 退出我们的命令提示符

环境变量：用于记录我们的的一些路径

为什么要配置环境变量：

当我们想去在任意的目录下打开制定的文件的时候，就可以吧软件的路径设置在环境变量之中

Java入门:

编译的理解：将代码转换成计算机可以识别的代码

Javac 编译我们的Java的源文件

然后通过Java运行我们编译后的文件

 image-20230426144007074

环境变量:

为什么配置我们的path的环境变量的呢?

其实我们就是为了让我们的Java中的Javac和Java这个写入到环境变量中，我们就可以在任何环境中都能够使用

Java能干嘛?

Javase: java语言的基础

javame:

用于嵌入式

Javaee:

用于网站的开发：

java跨平台：

高级语言的编译运行的方式：编程，编译，运行

编译性语言：C

无法跨平台：

解释性语言：python

混合性编译语言：JAVA

Java在运行的时候，编译的.class的文件是运行在虚拟机里面的，而不是运行在我们的系统上

 image-20230426145943448

JRE和JDK：

JDK的全称：Java开发工具包

1. Jvm java 虚拟机
2. 核心类库
3. 开发工具

   开发工具的构成：
4. javac开发工具
5. Java运行工具
6. Java调试工具
7. Java内存分析工具

JRE 是Java的运行环境

JDK 包含了 JRE

Java的基础语法：

注释：

单行注释

```
//注释
```

多行注释

```
/* 注释信息*/
```

文档注释

```
/** 注释信息 **/
```

```java
public class Main {
    //main方法，表示程序的主入口
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
    //作为输出的语句（打印我们的语句）
    //将其打印到我们的命令行
    /*java 的注释信息，我们将其保存在我们的文件中，
    */
}
```

关键词：

class：

用于创建或者定义一个类，而类是Java组成的最基本的组成单元

字面量：

字面量的类型

1. 整数类型: 1
2. 小数类型: 3.14
3. 字符串类型：“hello world”
4. 字符类型 : 'h'
5. 布尔类型: 只有俩个数值：true或者false
6. 空类型：null

```java
import java.util.TreeMap;
public class valuedemo1 {
    //这段代码主要是用于中变量的书写
    //整数
    public static void main(String[] args){
        //整数
        System.out.println(666);
        System.out.println(667);
        System.out.println(668);

        //小数
        System.out.println(1.93);
        System.out.println(2.12);
        System.out.println(3.16);

        //字符串
        System.out.println("黑马程序员");
        System.out.println("我是你爸爸");
        System.out.println("我是你爷爷");


        //字符
```

```
        System.out.println('h');
        System.out.println('x');
        System.out.println('z');

        //布尔
        System.out.println(true);
        System.out.println(false);
    }
}
```

特殊的符号：

```
\t 制表符：
在打印的时候，把前面的字符串的长度补齐到8，或者8 的整数倍，
```

变量：

```
数据类型 变量名字=数据值；
```

```
public class valuedemo2{
    //这段代码主要是用于中变量的书写
    //整数
    public static void main(String[] args){
        //变量：
        //s数据类型：限定我们的变量
        int x=1;
        int y=3;
        double z=3.16;
        String num="hello world";
        System.out.println(z);
        System.out.println(num);
    }
}
```

数据类型：

基本数据类型：

整数	**byte**	**-128---128**
	short	-32768-32768
	int	-2147483648-2147483648
	long	
浮点数	float	
	double	

整数	byte	-128---128
字符	char	0-65535
布尔变量	Boolean	Ture false

标识符：

硬性要求：

由数字和字母下划线和$构成

不能以数字起头

不能是关键字

区分大小写

软性的建议：

小驼峰命名法：方法变量

> 规范一：
> 标识符是一个单词的时候全部小写

> 规范2：
> 标识符有多个单词的时候，第一个单词的首字母小写起小雨的单词的首字母全部大写

大驼峰命名法：类名：

> 规范一：标识符是一个单词的时候，首字母大写

> 规范二：标识符有多个的单词构成的时候，单个单词的首字母需要大写

键盘的录入：

Java已经帮我写好scanner，这个类就可以接受键盘输入的数字

```
import  java.util.Scanner;
public class valuedemo4 {
    //这段代码主要是用于中变量的书写
    //整数
    public static void main(String[] args){
        //变量的键盘的输入
        //创建对象，表示我们要开始使用这个类
        Scanner sc=new Scanner(System.in);
        //接受数据-开始真正的干活
        System.out.println("请输入：");

        int i=sc.nextInt();
```

```
        System.out.println(i);

    }
}
```

运算符

```
package com.it.demo1;

public class demo2 {
    public static void main(String[] args) {
        //加法
        int a=10;
        int b=3;
        double x=10.0;
        System.out.println(a+b);

        //减法
        System.out.println(a-b);

        //乘法
        System.out.println(a*b);

        //除法

        System.out.println(a/b);

        System.out.println(x/b);

        //取余
        System.out.println(a%b);
    }
}
```

隐式转化和强制转换

隐式转换：自动类型提升，小范围转变为大范围

隐式转换的俩种提升的规则：

取值范围小的和取值范围大的，小的会一般先提升为大的，在运算

byte short char 三种类型的数据在运算的时候，都会直接提升为int,然后在进行运算

$$byte < short < int < long < float < double$$

强制转换：大范围到小范围；

将取值范围大的数值复制给取值范围小的变量，需要进行强制转换

格式：

```
目标类型名 变量名=（目标数据类型）被强制转换的类型
```

字符串运算：

字符串的“+”这个操作：
当“+”操作中出现字符串时，这歌”+“实际上是一个字符串的连接符，而不是运算符，会将前后的字符串香链接，产生新的字符串

连续的加的时候，从左到右逐个进行的；

自增自减运算：

```
package com.it.demo1;

public class demo4 {
    public static void main(String[] args) {
        int a=10;
        a++;
        System.out.println(a);
        ++a;
        System.out.println(a);
        a--;
        System.out.println(a);
        --a;
        System.out.println(a);
    }
}
```

参与运算：

```
package com.it.demo1;

public class demo4 {
    public static void main(String[] args) {

        int x=10;
        int z=10;
        int b=x++;
        int c=++z;
        System.out.println(b);
        System.out.println(c);
        System.out.println(x);
    }
}
```

输出：

```
b:10
c:11
x:11
```

自增自减的运算的时候：

放在后面，先用后加

放在前面，先加后减

赋值运算符：

运算后赋值

并且+=，-=，*=，/=，%=中都隐含了强制类型转换；

比如：

```
short s=1;
s+=1
//等价于s=s+1;
//平常都会先将s提升为int类型的
//然后int类型的s和1加
//再将结果复制给short s未发生报错。其实是隐含了强制类型转转；
s=short(s+1)
```

```
package com.it.demo1;

public class demo5 {
    public static void main(String[] args) {
        int a1=10;
        int b1=10;
        a1+=b1;
        System.out.println(a1);
        int a2=10;
        int b2=10;
        a2-=b2;
        System.out.println(a2);

        int a3=10;
        int b3=10;
        a3*=b3;
        System.out.println(a3);

        int a4=10;
        int b4=10;
        a4/=b4;
        System.out.println(a4);
```

```
        int a5=10;
        int b5=10;
        a5%=b5;
        System.out.println(a5);
    }
}
```

```
1:20
2:0
3:100
4:1
5:0
```

关系运算符：

符号	说明
==	判断是否相等
!=	判断是否不等
>	判断是否大于
<	判断是否小于
>=	判断是否大于等于
<=	判断是否小于等于

```
package com.it.demo1;

public class demo6 {
    public static void main(String[] args) {
        int a1=10;
        int b1=10;
        int c1=20;
        System.out.println(a1==b1);
        System.out.println(a1==c1);
        System.out.println(a1>=c1);
        System.out.println(a1<=c1);
    }
}
//true
//false
//false
//true
```

逻辑运算符：

符号	作用
&	逻辑与
|	逻辑或
^	逻辑异或
!	逻辑非

```
package com.it.demo1;

public class demo6 {
    public static void main(String[] args) {

        System.out.println(true & true);
        System.out.println(true | true);
        System.out.println(true ^ true);

    }
}
```

短路逻辑运算符：

符号	作用
&&（短路与）	前面的正确才可以运算后面的
||(短路或)	左边False 才会运行后面的，左边为True的时候直接结束

三元运算符：

三元运算符的格式：

格式：关系表达式？表达式1：表达式2；

实例：

```
c=a > b ? a:b;
```

```java
package com.it.demo1;

public class demo7 {
    public static void main(String[] args) {

        int a=10;
        int b=20;
        System.out.println(a>b?a:b);

    }
}
```

判断和循环：

```java
package com.it.demo1;

public class demo8 {
    public static void main(String[] args) {

        int a=10;
        int b=20;
        if (a>b){
            System.out.println("a>b");
        }
        else
            System.out.println("a<b");
    }
}
```

```java
package com.it.demo1;

public class demo8 {
    public static void main(String[] args) {

        int a=10;
        int b=20;
        if (a>b){
            System.out.println("a>b");
        } else if (a==b) {

            System.out.println("a==b");
        } else
            System.out.println("a<b");
    }
}
```

```java
package com.it.demo1;
```

```
public class swichdemo9 {
    public static void main(String[] args) {

        String noodle="兰州拉面";
        switch (noodle)
        {
            case "兰州拉面":
                System.out.println("兰州拉面");
                break;
            case "武汉热干面":
                System.out.println("武汉热干面");
                break;//break不能省略。不然后导致case穿透
            case "北京蘸酱面":
                System.out.println("北京蘸酱面");
                break;
            case "油泼面":
                System.out.println("油泼面");
                break;
            default:
                System.out.println("吃个屁");
                break;

        }
    }
}
//执行的流程
//首先还是会拿着我们的那表达式的值和我们的每一个case进行匹配
//如果匹配上，那么就会执行对应的预聚体，如果发现了break就会跳出
//如果没有发现break的时候，就会继续执行下一个的case的语句体
```

```
package com.it.demo1;

public class swichdemo9 {
    public static void main(String[] args) {
//
//        String noodle="兰州拉面";
//        switch (noodle)
//        {
//            case "兰州拉面":
//                System.out.println("兰州拉面");
//                break;
//            case "武汉热干面":
//                System.out.println("武汉热干面");
//                break;
//            case "北京蘸酱面":
//                System.out.println("北京蘸酱面");
//                break;
//            case "油泼面":
//                System.out.println("油泼面");
```

```
//                break;
//            default:
//                System.out.println("吃个屁");
//                break;
//
//        }
        int number=1;
        switch (number){
            case 1 -> System.out.println("1");
            case 2 -> System.out.println("2");
            case 3 -> System.out.println("3");
            case 4 -> System.out.println("4");
            default -> System.out.println("没有选项");

        }
    }
}
```

循环结构：

for循环

```
package com.it.demo1;

public class fordemo9 {
    public static void main(String[] args) {
     for (int i=0;i<10;i++){
         System.out.println(i);
     }
    }
}
```

```
package com.it.demo1;

public class fordemo9 {
    public static void main(String[] args) {
     for (int i=0;i<10;i++){
         System.out.println(i);
     }
        System.out.println("--------------------");
     for (int i=9;i>=0;i--){
         System.out.println(i);
     }
    }

}
```

累加的算法：

```
package com.it.demo1;

public class fordemo10 {
    public static void main(String[] args) {
        int sum=0;
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
            sum+=i;
        }
        System.out.println(sum);
    }
}
```

while:

```
package com.it.demo1;

public class whiledemo11 {
    public static void main(String[] args) {
        int i=0;
        while(i<10)
        {
            i++;
            System.out.println(i);
        }
    }
}
```

而二者循环的区别:

for循环中的控制循环的变量，归属于我们的for循环的语法结构，运行结束后，直接就不能再被访问

while循环，控制循环的变量定义在循环的外面因此在循环结束后可以继续访问shiyong

从开发的角度:

for循环一般是在我们知道循环的次数和循环的范围上使用

while循环；我们不知道循环的次数和范围，只知道循环的约束条件

力扣算法：回文数

```
package com.it.demo1;

public class whiledemo3 {
    //判断回文数:
    //如果一个x是回文整数，打印ture,否则打印False
    //解释：回文数指，正序和逆序都是一样的整数
    //例如121
    //把原来的数字倒过来和原始的做比较
```

```
    public static void main(String[] args) {

        int x=12321;
        int y=x;
        int num=0;
        //定义数字
        //不知道的时候采用while
        while(x!=0){
            int ge=x%10;
            x=x/10;
            num=num*10+ge;
        }
        System.out.println(num);
        if(y==num) {
            System.out.println("是回文数");
        }
        else
            System.out.println("不是回文数");
        }

    }
```

力扣算法：不用除法和取余数

```
package com.it.demo1;

public class whiledemo4 {
    //给定俩个整数，被除数和出书相除，要求不使用除法。得到商和余数

    public static void main(String[] args) {

        int a=201;
        int b=10;
        int count=0;
        while (a>b){

            a-=b;
            count++;

        }
        System.out.println(a);
        System.out.println(count);
    }


    }
```

无限循环：

三种形式：

```
for(;;){
}
```

```
while(true){

}
```

```
do{

}while(true)
```

跳转程序语句：

```
package com.it.demo1;

public class whiledemo6 {
    //给定俩个整数，被除数和出书相除，要求不使用除法。得到商和余数

    public static void main(String[] args) {

     for (int i=1;i<=5;i++){
         if(i==3)continue;
         System.out.println("小老鼠吃第几"+i+"个包子");
     }
    }
    }
```

输出：

```
小老鼠吃第几1个包子
小老鼠吃第几2个包子
小老鼠吃第几4个包子
小老鼠吃第几5个包子
```

```
package com.it.demo1;

public class whiledemo6 {

    public static void main(String[] args) {

     for (int i=1;i<=5;i++){
         if(i==3)break;
         System.out.println("小老鼠吃第几"+i+"个包子");
```

```
     }
    }
    }
```

输出：

```
小老鼠吃第几1个包子
小老鼠吃第几2个包子
```

七的倍数：

```
package com.it.demo1;

public class whiledemo7 {
    //逢7过
    //实现方法：1-100
    //包含7，个位或者十位包含7跳过
    //要求含有7 或者七得倍数的直接跳过

    public static void main(String[] args) {
    for(int i=1;i<=100;i++)
    {
        if(((i%10)==7)||((i/10%10)==7)||(i%7==0)){
            System.out.println("过");
            continue;
        }
        System.out.println(i);
    }

    }
    }
```

平方根：

```
package com.it.demo1;

public class whiledemo8 {
    //计算其输入的数的平方根并输出。
    //结果要求保留整数部分

    public static void main(String[] args) {
        int i=0;
        int count=0;
        int x=16;
        while (count<x) {
            count = i * i;
            if(count>x){
```

```
                System.out.println(i-1);
            } else if (count==x) {
                System.out.println(i);
            }
            else {
                i++;
            }

        }

    }
    }
```

判断是否为质数：

```
package com.it.demo1;

public class whiledemo9 {
    //判断是否为质数

    public static void main(String[] args) {

        int x=16;
        int count=2;
        while (count<=x-1) {
            if(x%count==0) {
                System.out.println("x=" + x + "是质数");
                break;
            }
            count++;
        }

    }
    }
```

```
package com.it.demo1;

public class whiledemo9 {
    //判断是否为质数

    public static void main(String[] args) {

        int x=16;
        int count=2;
        boolean flag=true;
        while (count<=x-1) {
            if(x%count==0) {
//                System.out.println("x=" + x + "是质数");
```

```
                flag=false;
                break;
            }
            count++;
        }
        if(flag==false){
            System.out.println("x=" + x + "不是质数");
        }
    }
    }
```

优化的判断质数：

数组：

数组的介绍：

数组是一种容器，存储同种数据类型的多个值；

数组容器在存储数据的时候需要结合隐式转换；

建议：容器的类型和存储的数据类型需要保持一致；

数组的定义和初始化：

```
格式：
数据类型 [] 数组名
格式：
数据类型 数组名[]
```

数组的初始化：

```
数据类型[] 数组名= new 数据类型[] {元素1，元素2，元素3}
```

实例：

```
package com.it.demo2;

public class demo1 {
    //数组的容器
    public static void main(String[] args) {
        int array[]=new int[]{11,12,13};
        double array2[]=new double[]{12.0,3.0,14.0};
        System.out.println(array[0]);
        ////简化的形式
        int array3[]={12,13,165,14};
        System.out.println(array3[1]);
    }
```

```
}
```

```
package com.it.demo2;

public class demo2 {
    //数组的容器
    public static void main(String[] args) {
        int array[]=new int[]{11,12,13};
        double array2[]=new double[]{12.0,3.0,14.0};
        String array3[]=new String[]{"张三","李四","王武"};
        for(int i=0;i<3;i++){
            System.out.println("------------------------");
            System.out.println("学生姓名："+array3[i]);
            System.out.println("学生年龄："+array[i]);
            System.out.println("学生身高："+array2[i]);
            System.out.println("------------------------");
        }
    }

}
```

数组的地址值和元素的访问：

```
数组的地址：
[I@2f92e0f4
//表示是一个数组
//I表示的是数据的类型
//2f92e0f4 数组的地址
```

数组的动态初始化：

初始化的时候只制定数组的长度，有系统为数组提供初始数值；

```
系统 的随机初始化：

int 默认制定为0

double 默认指定为0.0

字符类型 默认指定为空格

布尔类型 false
```

格式：数据类型 [] 数组名=new 数据类型[数组的长度]

```
package com.it.demo2;

public class demo3 {
    //数组的容器
    public static void main(String[] args) {
        int array[]=new int[]{11,12,13};
        int array4[]=new int[28];//动态初始化只需要指定数组的长度

        for(int i=0;i<28;i++){
            System.out.println("-------------------------");
            array4[i]=i;
            System.out.println(array4[i]);
            System.out.println("-------------------------");
        }
    }
}
```

方法:

简单的方法 的定义以及调用:

最简单的方法的定义:

带参数的方法的定义:

带有返回值的方法的定义:

```
package com.it.demo2;

public class demo3 {
    //数组的容器
    public static void main(String[] args) {
        int sum=0;
        MyFun();
        MyFun2(2);
        sum=MyFun3(3);
        System.out.println(sum);

    }
    public static void MyFun(){
        System.out.println("我的方法");
    }
    public static void MyFun2(int i){
        System.out.println("我的输入的数字"+i);
    }
    public static int MyFun3(int i){
        int sum=0;
        sum=i*2;
       return sum;
```

```
    }
}
```

方法的重载：

在同一个类中，定义了很多个同名的方法，这些tongi名字的方法具有同中的功能

每个方法具有不同的参数的类型和参数的个数，这些同名的方法就构成了重载的关系

```
package com.it.demo3;

public class demo3 {
    //数组的容器
    public static void main(String[] args) {
        int sum=0;
        int a=10;
        int b=10;
        int c=10;
        double x=1.0;
        double s=2.0;
        System.out.println(sum(a,b));
        System.out.println(sum(a,b,c));
        System.out.println(sum(x,s));
    }
    public static int sum(int a, int b){
        int sum=0;
        sum=a+b;
        return sum;
    }
    public static int sum(int a, int b,int c){
        int sum=0;
        sum=a+b+c;
        return sum;
    }
    public static double sum(double  a, double  b){
        double  sum=0;
        sum=a+b;
        return sum;
    }
}
```

数组的遍历：

```
package com.it.demo3;

public class demo2 {
    //数组的容器
    public static void main(String[] args) {
     int array[]=new int[100];
        for (int i = 0; i < array.length; i++) {
```

```
            array[i]=i;
        }
        sum(array);
    }

    public static void sum(int[] array) {
        for(int i=0;i< array.length;i++){
            System.out.println(array[i]);
        }


    }
}
```

寻找最大的值:

```
package com.it.demo3;

public class demo3 {
    //数组的容器
    public static void main(String[] args) {
     int array[]=new int[100];
     int max=0;
        for (int i = 0; i < array.length; i++) {
            array[i]=i;
        }
        max=Getmax(array);
        System.out.println(max);
    }

    public static int Getmax(int[] array) {
        int max= array[0];
        for(int i=0;i< array.length;i++){
           if(array[i]>max) {
               max = array[i];
           }


            }
        return max;
        }


    }
```

判断数组中是否存在一个数字

```
package com.itheima.test;

public class test {
    public static void main(String[] args) {
        int[] array={1,2,3,4,5,6,7,8,9,10};
        //判断是否在数组中存在一个数字

        System.out.println( constain(array,11));

    }
    //定义一个方法
    public static boolean constain(int[] array,int number){
        for (int i = 0; i < array.length; i++) {
            if (array[i]==number)
                return true;
        }
        //判断是否存在
        return false;
    }
}
```

这里需要提下return和break的区别所在：

return和循环不存在什么关系，和方法相关，是指结束方法之后返回结果；

break和switch关键字相关，或者结束循环

方法调用的基本内存原理;

首先JVM会在运行的时候占用一定的计算机的内存；

Java的内存分配：

Java内存分配主要是5个部分

栈；堆；方法区；本地方法栈；寄存器

（1）栈：方法运行时使用的内存，比如main方法运行，进入方法栈中执行

（2）堆：存储对象或者数组，new来创建的，都存储在堆内存

（3）方法区：存储可以运行的class文件（当加载一个类时，这个类的字节码文件就会被加载到方法区）

（4）本地方法栈：JVM在使用操作系统功能的时候使用，（与开发无关）

（5）寄存器：给CPU使用，（与开发无关）

方法调用的基本原理：先入后出的栈

方法调用的时候数据类型的传递：

基本数据类型：

数据值是存储在自己的空间中

特点：赋值给其他变量，赋的是真实的值。当b变量修改其本身的值后，a变量的值不发生改变。

引用数据类型：

数据值是存储在其他空间中的，自己空间中存储的是地址值。（本质是使用别人空间的东西）

特点：赋值给其他变量，赋的是地址值。当b变量修改其本身存储的地址所指向的内存中的值时，a变量本身存储的地址所指向的内存中的值也会发生改变，因为a和b变量存储的地址都指向同一块内存，指向的是同一个东西。

3. 方法传递数据的内存原理：

传递基本数据类型时，传递的是真实的数据，形参的改变，不影响实际参数的值

传递引用数据类型时，传递的是地址值，形参的改变，影响实际参数的值。

Java面向对象：

面向对象的介绍：

学习的内容：

学习获取已有的对象并且使用

学习自己设计对象并使用

类和对象：

类：设计图,是对对象共同的特征的描述

对象：是真实存在的具体东西

Java中必须先设计我们的类，再去设计具体的对象

如何定义类：

```
public class 类名{

成员变量(代表属性，一般为名词)
成员方法(代表行为，一般为动词)
构造器
代码块
内部类
}
```

举例：

```
public class phone(){
string brand;
double price;
public void call(){}
public void playgame(){}

}
```

首先定义我的对象：

```
package com.itheima.test;

public class Phone
{
   String brand;
   double price;

   public void call(){
       System.out.println("call my phone");

   }
    public void playgame() {
        System.out.println("play my game");
    }
}
```

调用定义得对象类以及里面的方法：

```
package com.itheima.test;

public class Phonetest {
    public static void main(String[] args) {
        //创建手机的对象
        Phone p=new Phone();

        //给手机幅值的操作
        p.brand="apple";
        p.price=1800;
        System.out.println(p.brand);
        System.out.println(p.price);


        //掉用手机中的方法v
        p.call();
        p.playgame();

    }
}
```

类和对象分别是什么呢?

首先我们定义的一个类；这个类其实是我们的共同特征的描述；

对象：是我们事实存在的具体的实例

类的注意事项：

定义类的具体的一些事项：

> 用来描述我们一类事物的类，专业的叫做：JavaBean类
>
> 在javabean类中，是一般不写main的方法的；
>
> 在之前的操作中 ，我们加入的main方法的类，叫做测试类
>
> 我们可以在测试类中创建javabean 类的对象，并进行赋值调用

> 一个Java文件中我们可以创建多个class,但是只能用一个类是用public来修饰，且public修饰的类必须作为我们的文件名

成员变量的命令格式；

基本类型：

short byte int long float double boolean

引用类型：

类 接口 数组 String

封装（重要）：

面向对象的三大特征之一：封装 继承 多态

对象代表什么，就得封装对应的数据，并提供数据所对应的行为

封装本质上就是告诉我们面向对象的设计：

怎么设计对象呢?

封装的好处：

便于我们快速的开发

private 关键词

是一个权限的修饰符

可以修饰成员

被private修饰的只可以在本类中才能访问

a:是一个权限修饰符

b:可以修饰成员变量和成员方法

c:被其修饰的成员只能在本类中被访问

private 私有的 是一个权限修饰符，可以修饰成员变量和成员方法，被修饰后，只能在本类中访问，外界无法访问。

public 公共的 是一个权限修饰符，可以修饰类，可以修饰成员变量，可以修饰成员方法，被修饰的成员在任何地方都可以访问

2.如果成员变量被private修饰了(私有化)那么我们怎么才可以在其他类中访问被修饰的成员变量呢？

我们可以借助set方法对成员变量赋值，get方法获得成员变量

举例：其实本质上就是在类中进行变量的操作，具体的实现就是使用我们定义对象的方法。

```
package com.itheima.test;

public class Girlfriend {
    private String name;
    private int age;
    private String gender;
    private double height;
    int weight;
    public void setName(String a){
            name=a;
    }
    public String getName(){
        return name;
    }
    public void setGender(String a){

        gender=a;

    }
    public String getGender(){
        return gender;
    }
    public void setHeight(double a){

        height=a;

    }
    public double getHeight(){
        return height;
    }
    public void set(int a){
        if(a>=18 && a<=50)
        {
```

```
            age=a;
        }
    }
    public int get(){
       return age;
    }
    public void sleep(){
        System.out.println("女朋友在睡觉");
    }

    public void eat() {
        System.out.println("女朋友在吃饭");

    }

    public void fight(){
        System.out.println("女朋友在格斗");
    }
}
```

```
package com.itheima.test;

public class Girlfriendtest {
    public static void main(String[] args) {
        Girlfriend p=new Girlfriend();
        p.setName("jiajia");
        p.setHeight(156);
        p.set(18);
        p.setGender("女");
        p.weight=120;
        System.out.println(p.getName());
        System.out.println(p.getHeight());
        System.out.println(p.get());
        System.out.println(p.getGender());
        System.out.println(p.weight);
        p.sleep();
        p.eat();
        p.fight();

        System.out.println("_____________________________________________________");

        Girlfriend q=new Girlfriend();
        q.setName("yaya");
        q.setHeight(176);
        q.set(20);
        q.setGender("女");
        q.weight=120;
        System.out.println(q.getName());
        System.out.println(q.getHeight());
        System.out.println(q.get());
```

```
        System.out.println(q.getGender());
        System.out.println(q.weight);
        q.sleep();
        q.eat();
        q.fight();


    }
}
```

成员变量和局部变量：

```
package com.itheima.test;

public class Girlfriends1 {

    private int age=100;

    public static void main(String[] args) {
      setAge();
    }
    public static void setAge() {
        int age = 10;
        System.out.println(age);
    }
}
代码的输出为10 而不是100?
原因： 就近原则
```

```
package com.itheima.test;

public class Girlfriends1 {

    private int age=100;
    public void setAge() {
        int age = 10;
        System.out.println(this.age);
    }
}
输出为100
```

this的特性：

1. this的类型：那个对象调用，就是哪个对象的引用类型
2. this只能在成员方法中使用
3. 在成员方法中，this只能引用当前对象，不能再引用其它对象，具有final属性
4. this是成员方法第一个隐藏的参数，编译器会自动传递
5. this不能为空！！！

$Java$中的$this$关键字表示对当前对象的引用，它用于区分实例变量和局部变量之间的命名冲突。

构造方法：

构造方法的格式：

```
public class student{

修饰符   类名（参数）{
方法体；
}
}
```

特点：

方法名字和类的名字是相同的，大小写也是一致的

没有返回值的类型

没有具体的返回的数值，不能又return带回结果数据

代码的实现

空参数构造：

```
package com.itheima.test;

public class student {
    private String name;
    private int age;

    public student(){
        System.out.println("看看执行了吗");
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```
package com.itheima.test;

public class student_test {
    public static void main(String[] args) {
        student x=new student();

    }

}
```

带参数的方法构造：

```
package com.itheima.test;
public class student {
    private String name;
    private int age;
    public student(String name,int age){
        this.name=name;
        this.age=age;

    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

```
package com.itheima.test;

public class student_test {
    public static void main(String[] args) {
        student x=new student("huangsan",12);
        System.out.println(x.getAge());
        System.out.println(x.getName());
    }

}
```

构造方法注意事项：

构造方法的定义：

没有定义构造方法，系统将会给出一个默认的无参数的构造方法

如果定义了构造方法，系统讲不在提供默认的构造的方法

构造方法的重载：

其实在定义的时候我们通常会把俩个都定义：

二者的方法名是一样的，但是参数不相同，叫做构造方法的重载

最推荐的方法：俩种构造的方法都一般会写上

标准JavaBean

标准的Javabean类

类名必须见名知意

成员变量使用private修饰

至少提供俩种构造的方法

成员的方法

提供每一个成员变量的set 和 get

> `JavaBean` 是一种符合特定规范的 `Java` 类，用于在 `Java` 应用程序中封装数据和操作。它是一种用于表示可重用组件的编程约定，通常用于实现数据传输对象（`Data Transfer Object`，`DTO`）或数据模型。
>
> `JavaBean` 类必须满足以下条件：
> 类必须是具有公共默认构造函数的具体类（不能是抽象类或接口）。
> 所有属性必须是私有的，并通过公共的 `getter` 和 `setter` 方法进行访问。
> 属性的命名必须遵循特定的命名规范（例如，属性 `"name"` 应该有对应的 `"getName()"` 和 `"setName()"` 方法）。
> 可以选择实现其他可选接口，如序列化接口（`Serializable`）。
>
> `JavaBean` 的主要目的是提供一种简单的方式来封装数据，使其可以轻松地进行访问和操作。通过使用 `getter` 和 `setter` 方法，可以控制对属性的访问，并在必要时执行数据验证、计算或其他逻辑操作。这种封装性使得 `JavaBean` 在各种应用程序中广泛应用，包括图形用户界面（`GUI`）开发、持久化数据存储、远程方法调用等。
>
> `JavaBean` 的设计原则是通过封装、私有化属性和提供公共的访问方法来实现封装和信息隐藏，同时提供了一种标准化的方式来创建可重用的组件，促进了面向对象编程的模块化和可维护性

对象内存图

引用数据类型：

数值存储在其他的空间中，自己的空间中存储的是我们的地址的数值

特点：赋值给其他的变量，赋值的是地址值。

基本数据类型：

数据值存储在自己的空间中。

特点：赋值给其他的变量，也是赋的真真实的数值

This关键字的内存原理：

this的本质其实是方法调用者的地址的数值。

成员和局部：

成员变量：类中的方法之外的变量

局部的变量：方法之中的变量

所在的位置不相同：

成员变量：堆内存

局部变量：栈内存

生命周期不相同：

成员变量：对象创建的时候而存在

局部变量：对象方法的消失而消失

作用域：

成员变量：整个类

局部变量：方法内

键盘输入的体系：

第一套体系：

nextInt():接受整数

nextDouble();接受小数

next():接受字符串

遇到空格，制表符，回车键停止接受，这些符号后面的东西不再接受

第二套新体系：

next Line():接受字符串

可以接受一整行的输入。

混合使用存在弊端

先用nextInt 后再使用nextLine，会出现nextLine接受nextInt没有接受的

API：

字符串的对象学习：

String类：

字符串在创建后是不可以更改的。

俩种赋值的方式：

直接赋值：

存储在串池中

可以检索相同的直接复用

节约内存

new出来：

存储在堆中，浪费内存

注意：

引用类型变量存储地址

基本数据类型存储的是数据的真实数值

字符串的比较：

`boolean`	**[equals]将此字符串与指定的对象比较。**
`boolean`	[equalsIgnoreCase]将此 `String` 与另一个 `String` 比较，不考虑大小写。

StringBuilder的使用：

可变的操作字符串的容器

可以高效的拼接字符串，还可以将字符串的内容反转

```
package com.itheima.test;
import java.util.Scanner;
public class demo1 {
    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
        System.out.println("请输入一个字符串：");

        String str=sc.next();
        StringBuilder sb=new StringBuilder(str);
        System.out.println(sb);
        System.out.println(sb.reverse());
        if (sb.toString()==sb.reverse().toString()){
            System.out.println(sb+"是对称字符串");
        }
        else {
            System.out.println(sb+"不是对称字符串");
        }
    }
}
```

StringJoiner的使用：

JDK8出现的，方便高效的拼接我们的字符串。

```
package com.itheima.test;
import java.util.StringJoiner;
//学习StringJoiner
//快速实现拼接
//StringJoiner sj=new StringJoiner("---","[","]");
//本质上第一个为中间的间隔符
//第二个为开头的第一个符号
//第三个为结束的符号
public class StringJoiner_demo1 {
    public static void main(String[] args) {
        StringJoiner sj=new StringJoiner("---","[","]");
        for (int i = 0; i < 10; i++) {
            String re="";
            re=re+i;
            sj.add(re);
        }
        System.out.println(sj.toString());
    }
}
```

字符串的原理

字符串存储的内存原理：

直接赋值的话会复用字符串常量池中的

new 出来的不会复用，而是会在堆中开辟出新的kongjian

字符串的==的比较的到底是什么

基本数据类型比较的时候，比较的是我们的实际的数值

比较引用的数据类型的时候，我们比较的是我们的地址的数值

字符串的拼接的原理

拼接的时候没有变量，都是字符串，出发字符串的优化机制，编译后就是最终

```
string s1="a"+"b"+"c";
其在编译的时候触发优化机制，直接处理成“abc”
```

拼接的时候有变量的时候，jdk8之前会使用string builder进行构建字符串，也就是使用一个变量和一个字符串相加，这个时候在我们的堆内存中会出现俩个对象，一个对象是我们的String Builder对象，一个是我们的字符串对象本身。

在JDK8之后，采用预估的机制，但是这种操作任然会出现浪费我们的空间，和时间。

我们应该怎么做?

后续的字符串的拼接一般不要使用我们的+ ，这个操作会直接在底层创建多个对象，浪费时间和性能。

一般采用String builder或者string joiner

String Builder的底层优化原理:

string builder是一个内容可变的容器;

集合

集合和数组的对比:

数组的长度是固定的

集合的长度是可变的

数组可以存储基本的数据类型，也可以存储引用数据类型

集合只能存储引用数据类型，要是想存储基本数据类型必须要把他们变成包装类

Arraylist:

每个 `ArrayList` 实例都有一个*容量*。该容量是指用来存储列表元素的数组的大小。它总是至少等于列表的大小。随着向 ArrayList 中不断添加元素，其容量也自动增长。并未指定增长策略的细节，因为这不只是添加元素会带来分摊固定时间开销那样简单。

```
package com.itheima.test;

import java.util.ArrayList;

public class ArrayList_demo1 {
    public static void main(String[] args) {
        ArrayList<String> al=new ArrayList<>();
        System.out.println(al);
    }

}
```

此时我们创建的是arraylist 的对象

这个类在我们的底层做了一些处理

打印对象的时候不是我们的地址值，是我们的集合中存储的数据的内容

打印的时候会吧数据用【】包裹;

对象中的方法：

增 add

增加元素，返回值表示是否添加成功

删 remove

删除指定的元素或者删除指定的数据元素的index

查 get

得到指定的索引的元素

改 set

修改指定的索引下的元素，返回原来的元素

example:

```
package com.itheima.test;
import java.util.ArrayList;
public class ArrayList_demo1 {
    public static void main(String[] args) {
        //创建一个集合
        ArrayList<String> al=new ArrayList<>();
        System.out.println(al);
        //添加元素
        al.add("aa");
        al.add("bb");
        al.add("cc");
        System.out.println(al);
        //改
        al.set(1,"ss");
        //删除
        al.remove("aa");
        al.add("aa");
        al.remove(1);
        System.out.println(al);


    }
}
```

基本数据创建集合：

基本数据类型创建集合的时候需要我们有对应的包装类

基本数据类型	**包装类**
byte	Byte
short	Short
char	**Char**

基本数据类型	包装类
int	**Integer**
long	Long
float	Float
double	Double
boolean	Boolean

```
package com.it.dem4;

import java.util.ArrayList;

public class learn_demo1 {
    public static void main(String[] args) {
        ArrayList<Integer> list=new  ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        System.out.println(list);
        int i=list.get(0);
        System.out.println(i);

    }
}
```

创建自己的对象数据的集合

```
package com.it.dem4;

import java.util.ArrayList;
import java.util.Scanner;

public class learn_demo3 {
    public static void main(String[] args) {
//定义一个集合添加学生对象，并遍历
        ArrayList<Student> stulist=new ArrayList<>();
        Scanner sr=new Scanner(System.in);

        for (int i = 0; i < 3 ; i++) {
            Student st=new Student();
            System.out.println("输入学生的姓名：");
            String name=sr.next();
            System.out.println("输入学生的年龄：");
            int age= sr.nextInt();
            System.out.println("输入学生的身高：");
            double height=sr.nextDouble();
```

```
            st.setName(name);
            st.setAge(age);
            st.setHeigh(height);
            System.out.println("---------------");
            stulist.add(st);
        }
        System.out.println("遍历所有的数据");
        for (int i = 0; i < stulist.size(); i++) {
            System.out.println("用户名字："+stulist.get(i).getName());
            System.out.println("年龄："+stulist.get(i).getAge());
            System.out.println("体重："+stulist.get(i).getHeigh());

        }
    }
}
```

注意；我们需要在这里考虑

```
 Student st=new Student();
```

这个她在循环的外面的时候就会导致我们的array list中存放的都是最后一个的对象的地址的内容。

所以这个定义的对象应该放在我们的循环的内部，每次new不同的地址。

查找用户：

```
package com.it.dem4;

public class User{
    private String id;
    private String username;
    private String password;

    public User() {
    }

    public User(String id, String username, String password) {
        this.id = id;
        this.username = username;
        this.password = password;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
```

```
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

主函数:

```
package com.it.dem4;

import java.util.ArrayList;

public class User_test {
    public static void main(String[] args) {
        ArrayList<User> user=new ArrayList<>();

        User u1=new User("1","001","1234");
        User u2=new User("2","002","1234");
        User u3=new User("3","003","1234");
        user.add(u1);
        user.add(u2);
        user.add(u3);
        System.out.println(contain(user,"4"));
        System.out.println(Mcontain(user,"1"));

    }
          public static Boolean contain(ArrayList<User> user,String id){
              for (int i = 0; i <  user.size() ;i++) {
                  String myid=user.get(i).getId();
                  if(myid.equals(id)){
                      return true;
                  }
              }
              return false;
          }
          public static int Mcontain(ArrayList<User> users,String id){
              for (int i = 0; i < users.size(); i++) {
               String myid=users.get(i).getId();
               if (myid.equals(id)){
                   return i;
               }
```

```
            }return -1;
        }
}
```

返回一系列的对象

```
package com.it.dem4;

import java.util.ArrayList;

public class Phone_test {
    public static void main(String[] args) {
        //创建集合对象
        ArrayList<Phone> ph=new ArrayList<>();
        Phone ph1=new Phone("小米",1000);
        Phone ph2=new Phone("苹果",3000);
        Phone ph3=new Phone("华为",4000);
        ph.add(ph1);
        ph.add(ph2);
        ph.add(ph3);
        ArrayList<Phone> mi=getPhone(ph);
        for (int i = 0; i < mi.size(); i++) {
            Phone ok=new Phone();
            ok=mi.get(i);
            System.out.println(ok.getBrand());
        }
    }

    //定义我自己的方法:
    public static ArrayList<Phone> getPhone(ArrayList<Phone> arrayList){
        ArrayList<Phone> my=new ArrayList<>();
        Phone myphone=new Phone();
        for (int i = 0; i < arrayList.size(); i++) {
            Phone p=arrayList.get(i);
            if(p.getPrice()>2000){
             myphone=p;
             my.add(myphone);
            }
        }

        return my;
    }
}
```

面向对象进阶：

static（静态）

static表示静态，是我们Java的一个修饰符，可以用来修饰我们成员方法和成员变量。

静态变量

表示呗static修饰的成员变量，叫做静态变量

特点：被该类的对象共享

调用方法：类名调用（推荐）

思考我们的静态变量是属于我们的类，并且可以被我们的类的对象所共用，因此我们采用类名调用

static的静态内存图

```
public class student{
string name;
int age
static string teachername;
pobulic void show(){
sout(name+age+teachername)
}
}
```

```
public class Test{
public void main(){
student.teachername="aweilaosi"
Student s1=new Student();
s1.name="zhangsan";
s1.age=23;
s1.show();
Student s2=new Student();
s2.show();
}
}
```

```
student.teachername="aweilaosi"
```

这个加载到我们的栈内存，这个时候会加载student的字节码文件加载到我们的方法区，在堆内存中创建了一个单独的静态存储的位置（静态区）

注意：静态变量是随着类的加载而加载的，优先于我们的对象出现。

```
Student s1=new Student();
```

该段代码的时候，会在我们的堆内存中创建我们的对象的空间

在这个空间中存储我们的其他的对象变量。

```
Student s2=new Student();
```

该段代码的时候，会在我们的堆内存中创建我们的对象的空间

在这个空间中存储我们的其他的对象变量

```
s2.show();
```

调用show的方法的时候，

show方法被加载入栈，找到对象的栈内存的中的内存的地址，然后找到静态区的变量，并全部展示出来

注意；静态区的的东西是我们类的，对象可以共享属性。

共享与否？静态与否

静态方法

表示被static修饰的成员方法，叫做静态方法。

特点：多用在我们的测试类和工具类中，Java bean中很少使用。

工具类：

帮助我们做一些事情但是不描述任何事物的类

规则：类名见名知意，

私有化我们的构造方法：不让外部创建对象，是因为这个工具类对象的创建对象不存在意义。

方法定义为静态的：方便共享。

Java bean类：

用来描述我们的一类事物的类。

测试类：

检查其他的类是否书写正确，带有main的方法，是我们程序的入口。

静态的关键字的注意事项：

```
静态方法只能访问静态变量
非静态方法可以访问静态和非静态的方法
静态方法中不能用this的关键字（结合内存的）
被static修饰的方法，其属于我们的类，可以直接使用，不需要new
没有static的方法，必须现创建对象。
```

```
public class MyClass {
    int num;
    static int age;
    public void method(){
        //成员方法既能访问静态变量也能访问成员变量
        System.out.println(num);
        System.out.println(age);
```

```
        System.out.println("这是个成员方法！");
}
    public static void methodStatic(){

        System.out.println(age);

     //   System.out.println(num);  错误。  静态方法只能访问静态变量
        System.out.println("这是个静态方法!");
       // 静态方法中不能使用this关键字
     //   System.out.println(this);
    }
}
```

```
/*
一旦使用static修饰成员方法，那么这就成为了静态方法。静态方法不属于对象，而是属于类的。
 如果没有static关键字，那么必须首先创建对象，然后通过对象才能使用它
  注意：静态只能直接访问静态。
       */
public class MyClass2 {
    public static void main(String[] args) {
        MyClass obj=new MyClass();  //首先创建对象
        //然后才能使用没有static关键字的内容
        obj.method();

        //对于静态方法来说，可以通过对象调用，也可以通过类调用
        obj.methodStatic();
        MyClass.methodStatic();


    }
}
```

内存的使用：

静态：

随着类的加载而加载

静态的东西就会在内存中互相使用，但是非静态的东西无法使用。

这个主要是因为在调用的时候，静态的随着类的加载而出现，此时的非静态的是不存在的，因此我们的静态的无法调用非静态的，但是静态的变量此时已经随着类的加载已经到静态存储区中了，因此是不影响他的使用的，所以静态方法无法调用实例的变量.

非静态：

非静态的可以调用所有的

继承

大量的对象出现内容的重复；

考虑：在这个上面进行递进的拓展。

父类：具有较为宽泛的属性和方法

子类：属性和方法更为具体。

Java中提供了一个关键字extends，用这个关键字，我们可以让一个类和另外一个类建立起来了继承的关系。

```
public class student extends Person{}
```

Student 成为子类或者派生类，person称为父类（基类或者超类）。

使用继承的好处：

可以把多个子类中重复的代码抽取到父类中，提高代码的复用性

继承学习的内容：

######

什么时候使用我们的继承?

当我们的类与类之间的时候，存在共性的内容，并且满足我们子类是父类的一种，就可以考虑使用我们的继承。

继承的特点：

Java只支持单继承，不支持多继承，但是支持多层的继承。

继承体系：

每一个类都直接或者间接的继承于object

Java中的所有的类都直接或者间接的继承于我们的object的类。

继承的变量访问：

在子类方法中 或者 通过子类对象访问成员时

如果访问的成员变量子类中有，优先访问自己的成员变量

如果访问的成员变量与父类中成员变量同名，则优先访问自己的.

```
成员变量访问遵循就近原则，自己有优先自己的，如果没有则向父类中找。
```

继承的时候的成员方法的访问：

成员方法的名字不相同的时候：

成员方法没有同名时，在子类方法中或者通过子类对象访问方法时，则优先访问自己的，自己没有时 再到父类中找，如果父类中也没有则报错。

成员方法的名字相同的时候：

(1)：通过子类对象访问父类与子类中不同名方法时，优先在子类中找，找到则访问，否则在父类中找，找到则访问，否则编译报错

（2）：通过派生类对象访问父类与子类同名方法时，如果父类和子类同名方法的参数列表不同(重载)，根据调用 方法适传递的参数选择合适的方法访问，如果没有则报错；如果父类和子类同名方法的原型一致(重写-后 面讲)，则只能访问到子类的，父类的无法通过派生类对象直接访问到。

super关键字

子类和父类中可能会存在相同名称的成员，如果要在子类方法中访问父类同名成 员时，该如何操作？直接访问是无法做到的，Java提供了super关键字，该关键字主要作用：在子类方法中访问父 类的成员（字段和方法）

```
public class Base {
int a;
int b;
public void methodA(){
System.out.println("Base中的methodA()");
}
public void methodB(){
System.out.println("Base中的methodB()");
}
}
public class Derived extends Base{
int a; // 与父类中成员变量同名且类型相同
char b; // 与父类中成员变量同名但类型不同
// 与父类中methodA()构成重载
public void methodA(int a) {
System.out.println("Derived中的method()方法");
}
// 与基类中methodB()构成重写(即原型一致，重写后序详细介绍)
public void methodB(){
System.out.println("Derived中的methodB()方法");
}
public void methodC(){
// 对于同名的成员变量，直接访问时，访问的都是子类的
a = 100; // 等价于： this.a = 100;
b = 101; // 等价于： this.b = 101;
// 注意：this是当前对象的引用
// 访问父类的成员变量时，需要借助super关键字
// super是获取到子类对象中从基类继承下来的部分
super.a = 200;
super.b = 201;
// 父类和子类中构成重载的方法，直接可以通过参数列表区分清访问父类还是子类方法
methodA(); // 没有传参，访问父类中的methodA()
methodA(20); // 传递int参数，访问子类中的methodA(int)
```

```
// 如果在子类中要访问重写的基类方法，则需要借助super关键字
methodB(); // 直接访问，则永远访问到的都是子类中的methodA()，基类的无法访问到
super.methodB(); // 访问基类的methodB()
}
}
```

注意：

1：只能在非静态方法中使用

2： 在子类方法中，访问父类的成员变量和方法

子类的构造方法；

子类不能直接继承我们父类的构造方法：

子类对象构造时，需要先调用基类构造方法，然后执行子类的构造方法

```
public class Base {
public Base(){
System.out.println("Base()");
}
}
public class Derived extends Base{
public Derived(){
// super(); // 注意子类构造方法中默认会调用基类的无参构造方法：super(),
// 用户没有写时,编译器会自动添加，而且super()必须是子类构造方法中第一条语句,
// 并且只能出现一次
System.out.println("Derived()");
}
}
public class Test {
public static void main(String[] args) {
Derived d = new Derived();
}
}
```

结果打印：

```
Base()
Derived()
```

在子类构造方法中，并没有写任何关于基类构造的代码，但是在构造子类对象时，先执行基类的构造方法，然后执 行子类的构造方法，因为：子类对象中成员是有两部分组成的，基类继承下来的以及子类新增加的部分 。父子父子 肯定是先有父再有子，所以在构造子类对象时候 ，先要调用基类的构造方法，将从基类继承下来的成员构造完整 ，然后再调用子类自己的构造方法，将子类自己新增加的成员初始化完整 。

注意：
1： 若父类显式定义无参或者默认的构造方法，在子类构造方法第一行默认有隐含的super()调用，即调用基类构造方法。

2： 如果父类构造方法是带有参数的，此时编译器不会再给子类生成默认的构造方法，此时需要用户为子类显式定义构造方法，并在子类构造方法中选择合适的父类构造方法调用，否则编译失败。//实践

3：在子类构造方法中，super(…)调用父类构造时，必须是子类构造函数中第一条语句。

4： super(…)只能在子类构造方法中出现一次，并且不能和this同时出现

this和super的区别：

相同点：
1： 只能在类的非静态方法中使用，用来访问非静态成员方法和字段

2： 在构造方法中调用时，必须是构造方法中的第一条语句，并且不能同时存在。

不同点：
1： this是当前对象的引用，当前对象即调用实例方法的对象，super相当于是子类对象中从父类继承下来部分成员的引用

2：在非静态成员方法中，this用来访问本类的方法和属性，super用来访问父类继承下来的方法和属性

3： this是非静态成员方法的一个隐藏参数，super不是隐藏的参数

4：在构造方法中：this(…)用于调用本类构造方法，super(…)用于调用父类构造方法，两种调用不能同时在构造方法中出现

子类继承的成员方法：

子类可以从父类继承一个虚方法表：

虚方法表： 非private 非 static 非 final的其他方法放入虚方法

多态：

什么是多态：

同种类型的对象，表现出来的不同的形态

多态的表现形式：

```
Person p=new teacher()
Person p=new student()
Person p=new administrator()
```

多态的前提：

有继承的关系

有父类引用指向子类的对象

```
package com.it.demo7;

import com.it.dem4.Student;

public class test {
    public static void main(String[] args) {
        student s=new student();
        s.setName("zhangsan");
        s.setAge(18);
        teacher t=new teacher();
        t.setName("zhangwei");
        t.setAge(29);
        Adminer a=new Adminer();
        a.setName("huangsan");
        a.setAge(39);
        register(s);
        register(t);
        register(a);

    }
    //这个方法可以接受老师学生管理员
    //怎么办
    //多态，我们可以使用他们的父类
  //这个传入的父类我们可以传入她的所有的子类
    public static void register(Person S){
        S.show();
    }
}
```

我们无论传入什么他都可以进行传入。在我们注册的时候特别有用。

多态调用我们成员的特点：

变量的调用：编译看左边，运行看左边

方法的调用：编译看左边，运行看右边

```
package com.it.demo7;

public class test {
    public static void main(String[] args) {
        student s=new student();
        s.setName("zhangsan");
        s.setAge(18);
        teacher t=new teacher();
        t.setName("zhangwei");
        t.setAge(29);
```

```
        Adminer a=new Adminer();
        a.setName("huangsan");
        a.setAge(39);
        register(s);
        register(t);
        register(a);
        //测试多态对我们的子类的处理
        Person ps=new student();
        //使用多态调用我们的变量的时候遵循我们的编译看左边，调用用右边
        System.out.println(ps.x);
        //10
        //测试多态对我们的方法的调用
        ps.show();
        //学生的信息：
        //使用多态的时候，遵循我们的编译看左边，调用看右边的基本的要求
        //多态的弊端
        student se=(student) ps;
        se.mys();

    }
    //这个方法可以接受老师学生管理员
    //怎么办
    //多态，我们可以使用他们的父类
    public static void register(Person S){
        S.show();
    }
}
```

为什么多态引用会有上述的要求?

我们可以这样思考，在使用这个多态的引用的时候：
  Person ps=new student();
现在用我们的Person类定义的时候，其实ps是我们的person的类，所以在找的时候会先在我们person中去找
  变量的调用：
  创建student的时候，其实我们的student中会继承我们父类的变量，此时的student中存在两个变量，这个时候我们会根据我们的ps的类种类区选择和他一类的变量
  方法的调用，其实本质会继承我们父类的方法，所以我们的子类其实是吧我们从父类继承的方法已经重写了，这个时候 再去调用的时候其实就是我们的子类的方法。

多态的优势和弊端

多态的优势：

在多态的形势下，右边的对象可以实现解开耦合，便于拓展

定义方法的时候，使用我们的父类作为参数的时候，我们可以接受所有的子类的对象，体现多态的拓展

多态的弊端：

调用不了我们的子类所专有的方法。（因为在编译的时候我们先去检查我们的父类的方法里面是否存在我们的需要调用的方法的，要是没有就会报错，这就是为什么我们无法调用我们的子类的专用的方法）

解决方法，继续变成我们原始的子类（类型强制转换）

```
        Person ps=new student();
        student se=(student) ps;
        se.mys();
```

多态的练习：

```
package com.it.demo8;

public class person {
    private String name;
    private int age;

    public person() {
    }

    public person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void keepPet(Animal s,String somethig){
        if(s instanceof dog){
        System.out.println("年龄为"+age+"的"+name+"养了一只"+s.getColor()+"的狗");
        s.eat(somethig);}
        else{
        System.out.println("年龄为"+age+"的"+name+"养了一只"+s.getColor()+"的猫");
        s.eat(somethig);
    }}
}
```

多态如何调用子类的专属的方法：

```
package com.it.demo8;

public class test {
    public static void main(String[] args) {
        person p=new person();
        cat c=new cat();
        c.setAge(2);
        c.setColor("yellow");
        dog d=new dog();
        d.setColor("red");
        d.setAge(12);
        p.setName("赵巍山");
        p.setAge(18);
        p.keepPet(c,"小鱼干");
        p.keepPet(d,"骨头");
        Animal x=c;
        if (x instanceof dog dd){
            dd.look_home();
        }
        else if(x instanceof cat dd) {
            dd.zhua();
        }
    }
}
```

面向对象中的细节：

包：包就是文件夹，用来管理不同功能的Java的类，方便我们的后期的管理；

包的命名的规则：公司的域名反写并加上包的作用，需要全部的英文的小写，简明之一

```
package com.it.demo8;
public class student{

}
```

我们在使用我们的具体的方法的时候要加上包的名字

使用其他的类的时候我们需要使用它的包名+对象名

之后的使用的时候

```
import com.it.demo7.student;
public static void main(){
student s1=new student;
}
```

Final关键词:

final修饰方法：表明我们的方法为最后的一个方法，不能重写。

final修饰一个类：表明这个类是最终的类，不能被继承。

final修饰变量：叫做常量，只能赋值一次。

常量:

常量的命名的规范：一般作为我们的系统的配置的信息，方便维护，提高我们的数据的可读性

单个单词：全部大写

多个单词：全部大写

细节：

final修饰的变量为基本的数据类型的时候，变量存书的数据值不能发生变化。

final修饰的引用的类型，那么变量存储的地址的数值不能发生改变，而对象的内部的可以gaibian

权限修饰符:

权限修饰符：用来控制每一个成员的能够被访问的权限范围。

修饰符	同一个类	同一个包下的类	不同的包下的类	不同包下的无关类	权限
private	ture				私有的
空着不写	ture	ture			只在本包中使用
Protected	ture	ture	ture		可以在本包和别包中使用
public	ture	ture	ture	ture	全部能使用

静态代码块:

```
格式:
static{}
```

特点：需要通过static的关键字修饰，随着类的加载而加载，并且自动的触发，只执行一次。

使用的场景：

在做类的加载，做数据的初始化的时候儿加载使用。

执行的时机：加载类的时候就会执行我们的静态的代码块

```
package com.it.demo8;
```

```
public class person {
    private String name;
    private int age;
    static {
        System.out.println("静态对象执行了");
    }
    public person() {
    }
    public person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void keepPet(Animal s,String somethig){
        if(s instanceof dog){
        System.out.println("年龄为"+age+"的"+name+"养了一只"+s.getColor()+"的狗");
        s.eat(somethig);}
        else{
        System.out.println("年龄为"+age+"的"+name+"养了一只"+s.getColor()+"的猫");
        s.eat(somethig);
    }}
}
```

抽象类：

abstract修饰的方法：

```
public Abstract
```

抽象方法：

将共性的行为和方法抽取到父类之后，由于每一个子类的执行的内容是不一样的，所以在父类中不能确定具体的方法体，该方法就是抽象方法体

定义：

```
public abstract 返回值的类型 方法名();
```

抽象类：

如果一个类中存在抽象的方法，那么该类就必须申明为抽象类

```
public Abstract class 类名{}
```

注意的事项：

抽象类不能实例化：

抽象类中不一定存在抽象方法，但是抽象方法所在的类一定是抽象类

抽象类可以有构造方法

抽象类中子类

1. 要么重写抽象类中的所有的抽象方法
2. 要么子类本身也是抽象类

接口：

```
Java 接口是一系列方法的声明，是一些方法特征的集合，一个接口只有方法的特征没有方法的实现。
```

接口中的定义了一系列的抽象的方法和常量；

实现接口其实本质就是去完成接口中所有的抽象方法；

什么是接口?

本质上讲，接口是一种特殊的抽象类，这种抽象类中只包含常量和方法的定义，而没有变量和方法的实现。

为什么需要接口：

接口就是为了解决Java单继承这个弊端而产生的，虽然一个类只能有一个直接父类，但是它可以实现多个接口，没有继承关系的类，也可以实现相同的接口。继承和接口的双重设计即保持了类的数据安全也变相实现了多继承。

特点：

方法和属性默认都是*public*修饰，也可以使用*protected*，但不能用private。
所有的属性都是静态的常量，默认省略了static和final修饰符，属性的值必须实例化（初始化）

接口的特点：

1. 用 interface 来定义。
2. 接口中的所有成员变量都默认是由public static final修饰的。
3. 接口中的所有方法都默认是由public abstract修饰的。
4. 接口没有构造方法。构造方法用于创建对象
5. 实现接口的类中必须提供接口中所有方法的具体实现内容。
6. 多个无关的类可以实现同一个接口
7. 一个类可以实现多个无关的接口

8. 与继承关系类似，接口与实现类之间存在多态性
9. 接口也可以继承另一个接口，使用extends关键字。
10. 实现接口的类中必须提供接口中所有方法的具体实现内容。
11. 多个无关的类可以实现同一个接口
12. 一个类可以实现多个无关的接口
13. 与继承关系类似，接口与实现类之间存在多态性

接口的定义和使用：

定义接口：

```
public interface Person {
    final String NAME = "我是Person接口中的常量";
    void walk();
    void run();
}
```

常量和抽象的方法；

接口比抽象类更加“抽象”，它下面不能拥有具体实现的方法，必须全部是抽象方法，所有的方法默认都是`public abstract`的，所以在接口主体中的方法，这两个修饰符无需显示指定。

接口除了`方法声明`外，还可以包含`常量声明`。在接口中定义的所有的常量都默认是`public`,`static`和`final`的。

接口中的成员声明不允许使用`private`和`protected`修饰符。

实例：

定义的接口：

```
package com.itheima.test4;

public interface swim {
    public abstract void Swim();
}
```

定义实现接口的类：

```
package com.itheima.test4;

public class wa extends Animal implements swim{
    public wa() {
    }

    public wa(String name, int age) {
        super(name, age);
    }
```

```
    @Override
    public void eat() {
        System.out.println("吃虫子");
    }

    @Override
    public void Swim() {
        System.out.println("蛙泳");
    }
}
```

如何访问内存分析工具：

首先程序不能结束：一般让咱们的程序卡在这个位置，一直不结束

```
package com.itheima.test4;


import java.util.Scanner;

public class TEST2 {
    public static void main(String[] args) {
        infer x=new infer();
        x.Swim();

        Scanner sc=new Scanner(System.in);
        sc.next();
    }
}
```

打开terminal的工具行：输入

```
jps
```

选择我们的测试程序的id

```
C:\Users\Administrator\IdeaProjects\mystudy>jps
3728 Launcher
5536 TEST2
13048 Jps
7676
```

接下来：

```
C:\Users\Administrator\IdeaProjects\mystudy>jhsdb hsdb
```

类的多接口的实现：

```
package com.itheima.test4;
public class inter_ac implements inter1,inter2,inter3{
    @Override
    public void meth1() {

    }
    @Override
    public void meth2() {
    }
    @Override
    public void meth3() {
    }
}
```

接口中出现重名发的-0方法的时候：只需要重写一次就可以了

```
package com.itheima.test4;

public interface inter1 {
    void  meth1();
}
package com.itheima.test4;

public interface inter3 {
    void  meth1();
}
```

```
package com.itheima.test4;

public abstract class inter_ac implements inter1,inter2,inter3 {
    @Override
    public void meth1() {

    }
    @Override
    public void meth2() {

    }
}
```

接口和接口之间的关系：

可以实现继承关系，此外牢记接口可以多继承：

```
package com.itheima.test5;

public interface inter3 extends inter1,inter2{
    void method3();
}
//接口实现 的多继承:
```

定义接口的实现类:

```
package com.itheima.test5;

public class inter_ar implements inter3{
    @Override
    public void method1() {

    }
    @Override
    public void method2() {
    }

    @Override
    public void method3() {

    }
}
```

实现的类中需要是西安我们的定义的的继承自inter1和inter2 的接口的全部的抽象方法;

内部类:

成员内部类:

写在成员位置的，属于外部类的成员

成员内部类可以被一些修饰符号所修饰，比如private，protected,public,static.

在成员的内部类中，Jdk16之前不能定义静态变量,JDK16开始才能够定义静态变量

变量的内存图:

```
public class Outer{
private int a=10;

class Inner{
private int a=20;


public void show(){
int a=30;
SOUT(a);  //30
SOUT(this.a);//20
SOUT(Outer.this.a);//10
```

```
}
}
}
```

静态内部类

注意静态的内部类中，只能访问外部类中的静态的变量和方法，如果需要访问非静态的成员需要我们先创建对象。

```
public class Outer{
private int a=10;

static class Inner{
private int a=20;
}
}
```

直接创建静态内部类的方式：

```
Outer.inner oi=new outer.inner();
```

如何定义我们的静态的内部类中的方法?

非静态的方法的调用：

```
先创建我们的对象，然后再去调用
```

静态的方法：

```
外部类名.内部的类名.方法名()
```

局部内部类

```
public class Outer{
private int a=10;
public void inetr(){
int a=10;
static class Inner{
private int a=20;
}
}
}
```

将我们的内部类定义在我们的方法里面的类，类似于方法里面的局部变量

外界无法直接的使用·需要我们在方法的内部创建对象并去使用

该类可以直接访问外部的类的成员，也可以访问方法内的具局部的变量

重点掌握：匿名内部类

匿名内部类，本质上就是内部隐藏了名字的内部类

匿名内部类的格式：

```
new 类名或者接口的名字{
重写方法
};
包含了继承或者实现
方法的重写
创建对象
整体上就是一个类的子类对象的或者接口的实现类对象
```

使用的场景：

当方法的参数是一个接口或者类的时候，

一接口为例子，可以传递这个接口的实现类对象

如果实现类只用一次，就可以用匿名内部类去简化代码

GUI

主界面的创建：

```
public class jfram extends JFrame {
    public void Gameframe(){
        this.setSize(520,520);
        this.setTitle("单机的小游戏");
        this.setAlwaysOnTop(true);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(WindowConstants.*EXIT_ON_CLOSE*);
        this.setLayout(null);
        //让界面显示出来，默认页面是不显示的
        this.setVisible(true);

    }
}[]()
```

菜单的制作：

```
        //创建Jmenubar
        JMenuBar jmenubar=new JMenuBar();
        jmenubar.setSize(520,10);
        //设置JMENU
        JMenu jmu1=new JMenu("功能");
        JMenu jmu2=new JMenu("关于");
        JMenu jmu0=new JMenu("0");
        //JMENU中添加对应的选项，Jmenuitem,此外需要注意的是这jmenu中可以嵌套jmenu，也就是说我们可以在选项的下面继续设置选项的设置。
```

```
        JMenuItem action0=new JMenuItem("0");
        JMenuItem action01=new JMenuItem("01");
        JMenuItem action1=new JMenuItem("1");
        JMenuItem action2=new JMenuItem("2");
        JMenuItem action3=new JMenuItem("3");
        JMenuItem action4=new JMenuItem("4");
        jmu0.add(action0);
        jmu0.add(action01);
        jmu1.add(jmu0);
        jmu1.add(action1);
        jmu1.add(action2);
        jmu1.add(action3);
        jmu1.add(action4);
        jmenubar.add(jmu1);
        jmenubar.add(jmu2);
//最后一步的设置其实是吧我们的jmenuBar 设置到我们的Jframe的主要界面之中去、
        this.setJMenuBar(jmenubar);
```

图片的添加：

ImageIcon:

描述图片的类，可以关连到我们计算机中的任意的类，但是一般会先把图片拷贝到当前的项目中

Jlabel :

管理文本和图片的类

```
//1，先对整个界面进行设置
    //取消内部居中放置方式
    this.setLayout(null);
//2，创建ImageIcon对象，并制定图片位置。
    ImageIcon imageIcon1 = new ImageIcon("image\\1.png");
//3，创建JLabel对象，并把ImageIcon对象放到小括号中。
    JLabel jLabel1 = new JLabel(imageIcon1);
//4，利用JLabel对象设置大小，宽高。
    jLabel1.setBounds(0, 0, 100, 100);
//5，将JLabel对象添加到整个界面当中。
    this.add(jLabel1);
```

事件：

常见的三种的监听的方式：

- 键盘监听 KeyListener

```
public class MyJFrame extends JFrame implements ActionListener {

    //创建一个按钮对象
    JButton jtb1 = new JButton("点我啊");
    //创建一个按钮对象
    JButton jtb2 = new JButton("再点我啊");
```

```
    public MyJFrame(){
        //设置界面的宽高
        this.setSize(603, 680);
        //设置界面的标题
        this.setTitle("拼图单机版 v1.0");
        //设置界面置顶
        this.setAlwaysOnTop(true);
        //设置界面居中
        this.setLocationRelativeTo(null);
        //设置关闭模式
        this.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        //取消默认的居中放置，只有取消了才会按照XY轴的形式添加组件
        this.setLayout(null);


        //给按钮设置位置和宽高
        jtb1.setBounds(0,0,100,50);
        //给按钮添加事件
        jtb1.addActionListener(this);


        //给按钮设置位置和宽高
        jtb2.setBounds(100,0,100,50);
        jtb2.addActionListener(this);


        //那按钮添加到整个界面当中
        this.getContentPane().add(jtb1);
        this.getContentPane().add(jtb2);

        //让整个界面显示出来
        this.setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        //对当前的按钮进行判断
        //获取当前被操作的那个按钮对象
        Object source = e.getSource();

        if(source == jtb1){
            jtb1.setSize(200,200);
        }else if(source == jtb2){
            Random r = new Random();
            jtb2.setLocation(r.nextInt(500),r.nextInt(500));
        }
    }
}
```

- 鼠标监听 MouseListener

  当设置了鼠标监听事件的时候，需要实现鼠标监听的接口

```
        //获取正确的验证码
        String codeStr = CodeUtil.getCode();
        Font rightCodeFont = new Font(null,1,15);
        //设置颜色
        rightCode.setForeground(Color.RED);
        //设置字体
        rightCode.setFont(rightCodeFont);
        //设置内容
        rightCode.setText(codeStr);
        //绑定鼠标事件
        rightCode.addMouseListener(this);
        //位置和宽高
        rightCode.setBounds(400, 133, 100, 30);
        //添加到界面
        this.getContentPane().add(rightCode);
```

  此时的

```
public class LoginJFrame extends JFrame implements MouseListener
    LoginJFrame属于这个鼠标监听的接口的实现类
```

  当同一个界面的实现类中出现同样的监听的事件的时候；我们需要先判断这个监听的事件属于哪一个事件，然后再去执行相应的操作

```
public void mousePressed(MouseEvent e) {
        Object st=e.getSource();
        if(st==x1){
            System.out.println("hello");
        }

    }
```

  同一个界面中的不同的组件有不一样的点击的事件的时候我们需要采用，判断点击事件的类型来实现相应的功能

```
    @Override
    public void mouseClicked(MouseEvent e) {
        Object source = e.getSource();
        if (source==rightCode){

        } else if (source==login) {

        } else if (source==register) {

        }

    }
```

- 动作监听 ActionListener

API:

math类:

```
public static int abs(int a)                    // 返回参数的绝对值
public static double ceil(double a)             // 返回大于或等于参数的最小整数
public static double floor(double a)            // 返回小于或等于参数的最大整数
public static int round(float a)                // 按照四舍五入返回最接近参数的int类型的值
public static int max(int a,int b)              // 获取两个int值中的较大值
public static int min(int a,int b)              // 获取两个int值中的较小值
public static double pow (double a,double b)    // 计算a的b次幂的值
public static double random()                   // 返回一个[0.0,1.0)的随机值
```

system类:

```
public static long currentTimeMillis()          // 获取当前时间所对应的毫秒值（当前时间为0时区所对应的时间即就是英国格林尼治天文台旧址所在位置）
public static void exit(int status)             // 终止当前正在运行的Java虚拟机，0表示正常退出，非零表示异常退出
public static native void arraycopy(Object src,  int  srcPos, Object dest, int destPos, int length); // 进行数值元素copy
```

runtime类

```
public static Runtime getRuntime()      //当前系统的运行环境对象
public void exit(int status)            //停止虚拟机
public int availableProcessors()        //获得CPU的线程数
public long maxMemory()                 //JVM能从系统中获取总内存大小（单位byte）
public long totalMemory()               //JVM已经从系统中获取总内存大小（单位byte）
public long freeMemory()                //JVM剩余内存大小（单位byte）
public Process exec(String command)     //运行cmd命令
```

Object 类

```
public String toString()                //返回该对象的字符串表示形式(可以看做是对象的内存地址值)
public boolean equals(Object obj)       //比较两个对象地址值是否相等；true表示相同，false表示不相同
protected Object clone()                //对象克隆
```

Objects类

Objects类中无无参构造方法，因此我们不能使用new关键字去创建Objects的对象。同时我们可以发现Objects类中所提供的方法都是静态的。因此我们可以通过类名直接去调用这些方法。

```
public static String toString(Object o)                    // 获取对象的字符串表现形式
public static boolean equals(Object a, Object b)           // 比较两个对象是否相等
public static boolean isNull(Object obj)                   // 判断对象是否为null
public static boolean nonNull(Object obj)                  // 判断对象是否不为null
```

BigInteger类

```
public BigInteger(int num, Random rnd)      //获取随机大整数，范围：[0 ~ 2的num次方-1]
public BigInteger(String val)               //获取指定的大整数
public BigInteger(String val, int radix)    //获取指定进制的大整数

下面这个不是构造，而是一个静态方法获取BigInteger对象
public static BigInteger valueOf(long val)  //静态方法获取BigInteger的对象，内部有优化
```

常见的成员方法

```
public BigInteger add(BigInteger val)                   //加法
public BigInteger subtract(BigInteger val)              //减法
public BigInteger multiply(BigInteger val)              //乘法
public BigInteger divide(BigInteger val)                //除法
public BigInteger[] divideAndRemainder(BigInteger val)   //除法，获取商和余数
public  boolean equals(Object x)                        //比较是否相同
public  BigInteger pow(int exponent)                    //次幂、次方
public  BigInteger max/min(BigInteger val)              //返回较大值/较小值
public  int intValue(BigInteger val)                    //转为int类型整数，超出范围数据有误
```

example:

```
package com.itheima.test7;

import java.math.BigInteger;

public class bigInteger_demo {
    public static void main(String[] args) {
        //字符串中的数字必须是整数
```

```
        //创建一个对象
        BigInteger big=new BigInteger("100",10);
        System.out.println(big);

        //可以不去构造而是静态的方法去获取一个biginteger的对象
        //细节：
        //表示的范围比较小，必须为long的范围之内，之外的就不能创建

        BigInteger big4=BigInteger.valueOf(123);
        System.out.println(big4);
        //for example
        //max is 9223372036854775807
        // 超这个范围就会出问题
        BigInteger big5=BigInteger.valueOf(9223372036854775807L);
        System.out.println(big5);

    }
}
```

BigDecimal类

```
public BigDecimal add(BigDecimal value)             // 加法运算
public BigDecimal subtract(BigDecimal value)        // 减法运算
public BigDecimal multiply(BigDecimal value)        // 乘法运算
public BigDecimal divide(BigDecimal value)          // 触发运算
```

除法：

```
public class BigDecimalDemo02 {

    public static void main(String[] args) {

        // 创建两个BigDecimal对象
        BigDecimal b1 = new BigDecimal("1") ;
        BigDecimal b2 = new BigDecimal("3") ;

        // 调用方法进行b1和b2的除法运算，并且将计算结果在控制台进行输出
        System.out.println(b1.divide(b2));

    }

}
```

数据存在一个问题就是这个会产生无限不循环的小数，这个就会出现报错，我们应该有个舍入的方法

```
BigDecimal divide(BigDecimal divisor, int scale, int roundingMode)
```

```
divisor:            除数对应的BigDecimal对象;
scale:              精确的位数;
roundingMode:       取舍模式;
```
取舍模式被封装到了RoundingMode这个枚举类中（关于枚举我们后期再做重点讲解），在这个枚举类中定义了很多种取舍方式。最常见的取舍方式有如下几个：
UP(直接进1) ， FLOOR(直接删除) ， HALF_UP(4舍五入),我们可以通过如下格式直接访问这些取舍模式：枚举类名.变量名

正则表达式：

正则表达式可以校验字符串是否满足一定的规则，并用来校验数据格式的合法性。

在文本中查找满足要求的内容。

校验字符串的时候是忒有用

```
public class Regexdemo1 {
    public static void main(String[] args) {
        String qq="12345678";
        System.out.println(checkqq(qq));
        System.out.println(qq.matches("[1-9]\\d{5,19}"));

    }
```

正则校验

```
qq.matches("[1-9]\\d{5,19}")
 //标识1-9 全部为数字 且范围再5-19
```

正则表达式中：

字符类

方括号标识一个范围：

1. \[abc\]：代表a或者b，或者c字符中的一个。
2. \[^abc\]：代表除a,b,c以外的任何字符。
3. [a-z]：代表a-z的所有小写字符中的一个。
4. [A-Z]：代表A-Z的所有大写字符中的一个。
5. [0-9]：代表0-9之间的某一个数字字符。
6. [a-zA-Z0-9]：代表a-z或者A-Z或者0-9之间的任意一个字符。
7. [a-dm-p]：a 到 d 或 m 到 p之间的任意一个字符。

逻辑运算符：

1. &&：并且
2. |　　：或者
3. \　：转义字符

预定义字 符

```
1. "." ： 匹配任何字符。
2. "\d"：任何数字[0-9]的简写；
3. "\D"：任何非数字\[^0-9\]的简写；
4. "\s"： 空白字符：[ \t\n\x0B\f\r] 的简写
5. "\S"： 非空白字符：\[^\s\] 的简写
6. "\w"：单词字符：[a-zA-Z_0-9]的简写
7. "\W"：非单词字符：\[^\w\]
```

数量词

```
X? X一次或者多次
X* X，0次或者多次
X+ X，一次或者多次
X{n} X,出现n次
X{n,} x至少n次
X{n,M} x 至少n次但是不超过M次
```

指定要求的正则的表达式：

要求可以看懂会改即可

忽略大小写的方式

```
//(?i) ：表示忽略后面数据的大小写
//忽略abc的大小写
String regex = "(?i)abc";
//a需要一模一样，忽略bc的大小写
String regex = "a(?i)bc";
//ac需要一模一样，忽略b的大小写
String regex = "a((?i)b)c";
```

本地的数据的爬取：

```
package com.itheima.a08regexdemo;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexDemo6 {
    public static void main(String[] args) {
        /* 有如下文本，请按照要求爬取数据。
                Java自从95年问世以来，经历了很多版本，目前企业中用的最多的是Java8和Java11，
                因为这两个是长期支持版本，下一个长期支持版本是Java17，相信在未来不久Java17也会逐
渐登上历史舞台
                要求:找出里面所有的JavaXX
```

```
         */

        String str = "Java自从95年问世以来，经历了很多版本，目前企业中用的最多的是Java8和Java11，" +
                "因为这两个是长期支持版本，下一个长期支持版本是Java17，相信在未来不久Java17也会逐渐登上历史舞台";


        //1.获取正则表达式的对象
        Pattern p = Pattern.compile("Java\\d{0,2}");
        //2.获取文本匹配器的对象
        //拿着m去读取str，找符合p规则的子串
        Matcher m = p.matcher(str);

        //3.利用循环获取
        while (m.find()) {
            String s = m.group();
            System.out.println(s);
        }


    }

    private static void method1(String str) {
        //Pattern:表示正则表达式
        //Matcher: 文本匹配器，作用按照正则表达式的规则去读取字符串，从头开始读取。
        //           在大串中去找符合匹配规则的子串。

        //获取正则表达式的对象
        Pattern p = Pattern.compile("Java\\d{0,2}");
        //获取文本匹配器的对象
        //m:文本匹配器的对象
        //str:大串
        //p:规则
        //m要在str中找符合p规则的小串
        Matcher m = p.matcher(str);

        //拿着文本匹配器从头开始读取，寻找是否有满足规则的子串
        //如果没有，方法返回false
        //如果有，返回true。在底层记录子串的起始索引和结束索引+1
        // 0,4
        boolean b = m.find();

        //方法底层会根据find方法记录的索引进行字符串的截取
        // substring(起始索引，结束索引);包头不包尾
        // (0,4)但是不包含4索引
        // 会把截取的小串进行返回。
        String s1 = m.group();
        System.out.println(s1);


        //第二次在调用find的时候，会继续读取后面的内容
```

```
        //读取到第二个满足要求的子串，方法会继续返回true
        //并把第二个子串的起始索引和结束索引+1，进行记录
        b = m.find();

        //第二次调用group方法的时候，会根据find方法记录的索引再次截取子串
        String s2 = m.group();
        System.out.println(s2);
    }
}
```

网络爬取数据：

```
public class RegexDemo7 {
    public static void main(String[] args) throws IOException {
        /* 扩展需求2:
            把连接:https://m.sengzan.com/jiaoyu/29104.html?ivk sa=1025883i
            中所有的身份证号码都爬取出来。
        */

        //创建一个URL对象
        URL url = new URL("https://m.sengzan.com/jiaoyu/29104.html?ivk
sa=1025883i");
        //连接上这个网址
        //细节:保证网络是畅通
        URLConnection conn = url.openConnection();//创建一个对象去读取网络中的数据
        BufferedReader br = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
        String line;
        //获取正则表达式的对象pattern
        String regex = "[1-9]\\d{17}";
        Pattern pattern = Pattern.compile(regex);//在读取的时候每次读一整行
        while ((line = br.readLine()) != null) {
            //拿着文本匹配器的对象matcher按照pattern的规则去读取当前的这一行信息
            Matcher matcher = pattern.matcher(line);
            while (matcher.find()) {
                System.out.println(matcher.group());
            }
        }
        br.close();
    }
}
```

贪婪爬取和非贪婪爬取

只写+和表示贪婪匹配，如果在+和后面加问号表示非贪婪爬取
+? 非贪婪匹配
*? 非贪婪匹配
贪婪爬取:在爬取数据的时候尽可能的多获取数据
非贪婪爬取:在爬取数据的时候尽可能的少获取数据

举例：
如果获取数据：ab+
贪婪爬取获取结果:abbbbbbbbbbbb
非贪婪爬取获取结果:ab

```
/*
            有一段字符串:小诗诗dqwefqwfqwfwq12312小丹丹dqwefqwfqwfwq12312小惠惠
            要求1:把字符串中三个姓名之间的字母替换为vs
            要求2:把字符串中的三个姓名切割出来*/

String s = "小诗诗dqwefqwfqwfwq12312小丹丹dqwefqwfqwfwq12312小惠惠";
//细节:
//方法在底层跟之前一样也会创建文本解析器的对象
//然后从头开始去读取字符串中的内容，只要有满足的，那么就切割。
String[] arr = s.split("[\\w&&[^_]]+");
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
```

```
/*
            有一段字符串:小诗诗dqwefqwfqwfwq12312小丹丹dqwefqwfqwfwq12312小惠惠
            要求1:把字符串中三个姓名之间的字母替换为vs
            要求2:把字符串中的三个姓名切割出来*/

String s = "小诗诗dqwefqwfqwfwq12312小丹丹dqwefqwfqwfwq12312小惠惠";
//细节:
//方法在底层跟之前一样也会创建文本解析器的对象
//然后从头开始去读取字符串中的内容，只要有满足的，那么就用第一个参数去替换。
String result1 = s.replaceAll("[\\w&&[^_]]+", "vs");
System.out.println(result1);
```

正则表达中的分组的括号:

只看左括号，不看右括号，按照左括号的顺序，从左往右，依次为第一组，第二组，第三组等等

```
//需求1:判断一个字符串的开始字符和结束字符是否一致?只考虑一个字符
//举例: a123a b456b 17891 &abc& a123b(false)
// \\组号:表示把第X组的内容再出来用一次
String regex1 = "(.).+\\1";
System.out.println("a123a".matches(regex1));
System.out.println("b456b".matches(regex1));
System.out.println("17891".matches(regex1));
```

```
System.out.println("&abc&".matches(regex1));
System.out.println("a123b".matches(regex1));
System.out.println("--------------------------");


//需求2:判断一个字符串的开始部分和结束部分是否一致?可以有多个字符
//举例: abc123abc b456b 123789123 &!@abc&!@ abc123abd(false)
String regex2 = "(.+).+\\1";
System.out.println("abc123abc".matches(regex2));
System.out.println("b456b".matches(regex2));
System.out.println("123789123".matches(regex2));
System.out.println("&!@abc&!@".matches(regex2));
System.out.println("abc123abd".matches(regex2));
System.out.println("----------------------");

//需求3:判断一个字符串的开始部分和结束部分是否一致?开始部分内部每个字符也需要一致
//举例: aaa123aaa bbb456bbb 111789111 &&abc&&
//(.):把首字母看做一组
// \\2:把首字母拿出来再次使用
// *:作用于\\2,表示后面重复的内容出现日次或多次
String regex3 = "((.)\\2*).+\\1";
System.out.println("aaa123aaa".matches(regex3));
System.out.println("bbb456bbb".matches(regex3));
System.out.println("111789111".matches(regex3));
System.out.println("&&abc&&".matches(regex3));
System.out.println("aaa123aab".matches(regex3));
```

分组的替换

```
String str = "我要学学编编编编程程程程程程";
//需求:把重复的内容 替换为 单个的
//学学                  学
//编编编编              编
//程程程程程程          程
//  (.)表示把重复内容的第一个字符看做一组
//  \\1表示第一字符再次出现
//  + 至少一次
//  $1 表示把正则表达式中第一组的内容，再拿出来用
String result = str.replaceAll("(.)\\1+", "$1");
System.out.println(result);
```

时间相关类的学习

Date

时间类，用来描述我们的时间，精确到我们呢的毫秒

利用空参数构造创建的对象，默认标识系统的当前时间

利用有参数的构造默认为指定的时间

对象的创建

```
Date()
          分配 Date 对象并初始化此对象，以表示分配它的时间（精确到毫秒）。

Date data=new Date();表示现在的系统的当前的时间
Date data=new Date(指定的毫秒的数值); 表示指定的时间
```

修改时间对象中的毫秒数值

```
Date data=new Date(指定的毫秒的数值); 表示指定的时间
data.setTime(毫秒设置)
```

获取时间对象中的毫秒的数值

```
Date data=new Date(指定的毫秒的数值); 表示指定的时间
data.getTime()
```

SimpleDataFormat

格式化时间的类

解析，将字符串的时间转成我们的date的对象

定义的字符：

```
y  年  Year  1996; 96
M  年中的月份  Month  July; Jul; 07
w  年中的周数  Number  27
W  月份中的周数  Number  2
D  年中的天数  Number  189
d  月份中的天数  Number  10
F  月份中的星期  Number  2
E  星期中的天数  Text  Tuesday; Tue
a  Am/pm 标记  Text  PM
H  一天中的小时数（0-23）  Number  0
k  一天中的小时数（1-24）  Number  24
K  am/pm 中的小时数（0-11）  Number  0
h  am/pm 中的小时数（1-12）  Number  12
m  小时中的分钟数  Number  30
s  分钟中的秒数  Number  55
S  毫秒数  Number  978
z  时区  General time zone  Pacific Standard Time; PST; GMT-08:00
Z  时区  RFC 822 time zone  -0800
```

构造的方法：

```
SimpleDateFormat()
          用默认的模式和默认语言环境的日期格式符号构造
SimpleDateFormat(String pattern)
          用给定的模式和默认语言环境的日期格式符号构造
SimpleDateFormat(String pattern, DateFormatSymbols formatSymbols)
          用给定的模式和日期符号构造 SimpleDateFormat。
SimpleDateFormat(String pattern, Locale locale)
          用给定的模式和给定语言环境的默认日期格式符号构造
```

格式化的方法：

```
Thu Jan 01 08:00:00 CST 1970package com.itheima.test9;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.logging.SimpleFormatter;

public class Timetest1 {
    public static void main(String[] args) {
        SimpleDateFormat sd=new SimpleDateFormat();
        Date dt=new Date(0L);
        System.out.println(dt);
        System.out.println(sd.format(dt));
    }
}
输出：
1970/1/1 上午8:00
```

```
public class Timetest1 {
    public static void main(String[] args) {
        SimpleDateFormat sd=new SimpleDateFormat();
        Date dt=new Date(0L);
        System.out.println(dt);
        System.out.println(sd.format(dt));
        //带参数的构造的方法
        SimpleDateFormat sd1=new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss");
        System.out.println(sd1.format(dt));
    }
}
原格式：
Thu Jan 01 08:00:00 CST 1970
输出：
1970年01月01日 08:00:00
```

```
public class Timetest1 {
    public static void main(String[] args) throws ParseException {
        SimpleDateFormat sd=new SimpleDateFormat();
```

```
        Date dt=new Date(0L);
        System.out.println(dt);
        System.out.println(sd.format(dt));
        //带参数的构造的方法
        SimpleDateFormat sd1=new SimpleDateFormat("yyyy年MM月dd日 HH时:mm分:ss秒 E");
        System.out.println(sd1.format(dt));

        //解析时间的类
        String str="2022-12-12 12:12:12";
        SimpleDateFormat sd2=new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date mydate=sd2.parse(str);
        System.out.println(mydate.getTime());

    }
}
输出:
Thu Jan 01 08:00:00 CST 1970
1970/1/1 上午8:00
1970年01月01日 08时:00分:00秒 周四
1670818332000
```

实现解析后转换成指定的格式:

```
public class Timetest2 {
    public static void main(String[] args) throws ParseException {
        //带参数的构造的方法
        SimpleDateFormat sd1=new SimpleDateFormat("yyyy年MM月dd日");
        //解析时间的类
        String str="2022-12-12";
        SimpleDateFormat sd2=new SimpleDateFormat("yyyy-MM-dd");
        Date mydate=sd2.parse(str);
        System.out.println(mydate.getTime());
        System.out.println(sd1.format(mydate.getTime()));

    }
}
```

Calendar

代表系统当前的时间的日历的对象，可以单独的修改，获取时间中的年月日。

构造方法

```
protected  Calendar() 构造一个带有默认时区和语言环境的 Calendar。
protected  Calendar(TimeZone zone, Locale aLocale) 构造一个带有指定时区和语言环境的
Calendar
```

细节:他是一抽象类不能用来直接创建对象

日历对象的创建：

```
Calendar c=Calendar.getInstance();创建一个日历的对象获取到当前的时区的信息
```

日历的创建

```
        Calendar c=Calendar.getInstance();
        System.out.println(c);
        System.out.println(c.getTime());
        Date d=new Date(0L);
        c.setTime(d);
        System.out.println(c.getTime());
```

修改和获取指定的对象：

```
//get 获取的是日期中某个字段的信息，
        //set 修改的日历中某一个字段的信息
        //add 为某一个字段增加/减少指定的数值
        //0纪元
        //1年
        //2月
        //3一年中的第几周
        //4一个月中的第几周
        //5一个月中的第几天
        Calendar c1=Calendar.getInstance();
        System.out.println(c1.get(Calendar.YEAR));
        System.out.println(c1.get(Calendar.MONTH)+1);
        System.out.println(c1.get(Calendar.DATE));
        System.out.println(c1.get(Calendar.DAY_OF_WEEK));
        c1.set(Calendar.YEAR,1997);
        System.out.println(c1.getTime());
        c1.add(Calendar.MONTH,2);
        System.out.println(c1.getTime());
```

完整代码：

```
public class Timetest4 {
    public static void main(String[] args) throws ParseException {
        //Calendar 是一个抽象类，不能直接的new,而是通过一个静态的方法获取到我们的子对象，
        //个根据系统的时区获取不同的日历的对象
        //创建的会把时间中的年月日时分秒星期等存储在一个数组之中
        //细节2
        //星期在老外眼里是一周中的第一天
        Calendar c=Calendar.getInstance();
        System.out.println(c);
        System.out.println(c.getTime());
        Date d=new Date(0L);
        c.setTime(d);
        System.out.println(c.getTime());
        //get 获取的是日期中某个字段的信息，
```

```
        //set 修改的日历中某一个字段的信息
        //add 为某一个字段增加/减少指定的数值
        //0纪元
        //1年
        //2月
        //3一年中的第几周
        //4一个月中的第几周
        //5一个月中的第几天
        Calendar c1=Calendar.getInstance();
        System.out.println(c1.get(Calendar.YEAR));
        System.out.println(c1.get(Calendar.MONTH)+1);
        System.out.println(c1.get(Calendar.DATE));
        System.out.println(c1.get(Calendar.DAY_OF_WEEK));
        c1.set(Calendar.YEAR,1997);
        System.out.println(c1.getTime());
        c1.add(Calendar.MONTH,2);
        System.out.println(c1.getTime());
    }
}
```

JDK8 中新增时间相关类

JDK8时间类类名	作用
ZoneId	时区
Instant	时间戳
ZoneDateTime	带时区的时间
DateTimeFormatter	用于时间的格式化和解析
LocalDate	年、月、日
LocalTime	时、分、秒
LocalDateTime	年、月、日、时、分、秒
Duration	时间间隔（秒，纳，秒）
Period	时间间隔（年，月，日）
ChronoUnit	时间间隔（所有单位）

ZoneId时区

```
public class Timetest5 {
    public static void main(String[] args) throws ParseException {
      /*
      static Set<String> getAvailableZoneIds()  获取java 支持的时区
      static ZoneId systemDefault()             获取系统默认的时区
      static ZoneId of(String zoneId)           获取一个指定的时区
       */
```

```
        Set<String> availableZoneIds = ZoneId.getAvailableZoneIds();//获取所有的时区的名称
        System.out.println(availableZoneIds.size());
        ZoneId zoneId = ZoneId.systemDefault(); //获取当前的系统的默认的时区
        System.out.println(zoneId);
        System.out.println(ZoneId.of("Asia/Singapore"));//获取指定的时区

    }
}
```

Instant

常见的方法：

```
            static Instant now() 获取当前时间的Instant对象(标准时间)
            static Instant ofXxxx(long epochMilli) 根据(秒/毫秒/纳秒)获取Instant对象
            ZonedDateTime atZone(ZoneIdzone) 指定时区
            boolean isxxx(Instant otherInstant) 判断系列的方法
            Instant minusXxx(long millisToSubtract) 减少时间系列的方法
            Instant plusXxx(long millisToSubtract) 增加时间系列的方法
```

完整的代码：

```
package com.itheima.test9;

import java.text.ParseException;
import java.time.Instant;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.Calendar;
import java.util.Date;
import java.util.Set;

public class Timetest5 {
    public static void main(String[] args) throws ParseException {
     /*
            static Instant now() 获取当前时间的Instant对象(标准时间)
            static Instant ofXxxx(long epochMilli) 根据(秒/毫秒/纳秒)获取Instant对象
            ZonedDateTime atZone(ZoneIdzone) 指定时区
            boolean isxxx(Instant otherInstant) 判断系列的方法
            Instant minusXxx(long millisToSubtract) 减少时间系列的方法
            Instant plusXxx(long millisToSubtract) 增加时间系列的方法
        */
//1.获取当前时间的Instant对象(标准时间)
        Instant now = Instant.now();
        System.out.println(now);

//2.根据(秒/毫秒/纳秒)获取Instant对象
        Instant instant1 = Instant.ofEpochMilli(0L);
        System.out.println(instant1);//1970-01-01T00:00:00z
```

```
        Instant instant2 = Instant.ofEpochSecond(1L);
        System.out.println(instant2);//1970-01-01T00:00:01Z

        Instant instant3 = Instant.ofEpochSecond(1L, 1000000000L);
        System.out.println(instant3);//1970-01-01T00:00:027

//3. 指定时区
        ZonedDateTime time = Instant.now().atZone(ZoneId.of("Asia/Shanghai"));
        System.out.println(time);

//4.isXxx 判断
        Instant instant4=Instant.ofEpochMilli(0L);
        Instant instant5 =Instant.ofEpochMilli(1000L);

//5.用于时间的判断
//isBefore:判断调用者代表的时间是否在参数表示时间的前面
        boolean result1=instant4.isBefore(instant5);
        System.out.println(result1);//true

//isAfter:判断调用者代表的时间是否在参数表示时间的后面
        boolean result2 = instant4.isAfter(instant5);
        System.out.println(result2);//false

//6.Instant minusXxx(long millisToSubtract) 减少时间系列的方法
        Instant instant6 =Instant.ofEpochMilli(3000L);
        System.out.println(instant6);//1970-01-01T00:00:03Z

        Instant instant7 =instant6.minusSeconds(1);
        System.out.println(instant7);//1970-01-01T00:00:02Z
    }
}
```

ZoneDateTime 带时区的时间

```
/*
            static ZonedDateTime now() 获取当前时间的ZonedDateTime对象
            static ZonedDateTime ofXxxx(。。。) 获取指定时间的ZonedDateTime对象
            ZonedDateTime withXxx(时间) 修改时间系列的方法
            ZonedDateTime minusXxx(时间) 减少时间系列的方法
            ZonedDateTime plusXxx(时间) 增加时间系列的方法
         */
//1.获取当前时间对象(带时区)
ZonedDateTime now = ZonedDateTime.now();
System.out.println(now);

//2.获取指定的时间对象(带时区)1/年月日时分秒纳秒方式指定
ZonedDateTime time1 = ZonedDateTime.of(2023, 10, 1,
                                       11, 12, 12, 0, ZoneId.of("Asia/Shanghai"));
System.out.println(time1);

//通过Instant + 时区的方式指定获取时间对象
```

```
Instant instant = Instant.ofEpochMilli(0L);
ZoneId zoneId = ZoneId.of("Asia/Shanghai");
ZonedDateTime time2 = ZonedDateTime.ofInstant(instant, zoneId);
System.out.println(time2);


//3.withXxx 修改时间系列的方法
ZonedDateTime time3 = time2.withYear(2000);
System.out.println(time3);

//4. 减少时间
ZonedDateTime time4 = time3.minusYears(1);
System.out.println(time4);

//5.增加时间
ZonedDateTime time5 = time4.plusYears(1);
System.out.println(time5);
```

DateTimeFormatter 用于时间的格式化和解析

```
/*
            static DateTimeFormatter ofPattern(格式) 获取格式对象
            String format(时间对象) 按照指定方式格式化
        */
//获取时间对象
ZonedDateTime time = Instant.now().atZone(ZoneId.of("Asia/Shanghai"));

// 解析/格式化器
DateTimeFormatter dtf1=DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm;ss EE a");
// 格式化
System.out.println(dtf1.format(time));
```

LocalDate 年、月、日

```
//1.获取当前时间的日历对象(包含 年月日)
LocalDate nowDate = LocalDate.now();
//System.out.println("今天的日期:" + nowDate);
//2.获取指定的时间的日历对象
LocalDate ldDate = LocalDate.of(2023, 1, 1);
System.out.println("指定日期:" + ldDate);

System.out.println("==============================");

//3.get系列方法获取日历中的每一个属性值//获取年
int year = ldDate.getYear();
System.out.println("year: " + year);
//获取月//方式一:
Month m = ldDate.getMonth();
System.out.println(m);
System.out.println(m.getValue());
```

```
//方式二:
int month = ldDate.getMonthValue();
System.out.println("month: " + month);


//获取日
int day = ldDate.getDayOfMonth();
System.out.println("day:" + day);

//获取一年的第几天
int dayofYear = ldDate.getDayOfYear();
System.out.println("dayOfYear:" + dayofYear);

//获取星期
DayOfWeek dayOfWeek = ldDate.getDayOfWeek();
System.out.println(dayOfWeek);
System.out.println(dayOfWeek.getValue());

//is开头的方法表示判断
System.out.println(ldDate.isBefore(ldDate));
System.out.println(ldDate.isAfter(ldDate));

//with开头的方法表示修改，只能修改年月日
LocalDate withLocalDate = ldDate.withYear(2000);
System.out.println(withLocalDate);

//minus开头的方法表示减少，只能减少年月日
LocalDate minusLocalDate = ldDate.minusYears(1);
System.out.println(minusLocalDate);


//plus开头的方法表示增加，只能增加年月日
LocalDate plusLocalDate = ldDate.plusDays(1);
System.out.println(plusLocalDate);

//-------------
// 判断今天是否是你的生日
LocalDate birDate = LocalDate.of(2000, 1, 1);
LocalDate nowDate1 = LocalDate.now();

MonthDay birMd = MonthDay.of(birDate.getMonthValue(), birDate.getDayOfMonth());
MonthDay nowMd = MonthDay.from(nowDate1);

System.out.println("今天是你的生日吗? " + birMd.equals(nowMd));//今天是你的生日吗?
```

LocalTime 时、分、秒

```
// 获取本地时间的日历对象。(包含 时分秒)
LocalTime nowTime = LocalTime.now();
System.out.println("今天的时间:" + nowTime);
int hour = nowTime.getHour();//时
System.out.println("hour: " + hour);
```

```
int minute = nowTime.getMinute();//分
System.out.println("minute: " + minute);
int second = nowTime.getSecond();//秒
System.out.println("second:" + second);
int nano = nowTime.getNano();//纳秒
System.out.println("nano:" + nano);
System.out.println("-----------------------------------");
System.out.println(LocalTime.of(8, 20));//时分
System.out.println(LocalTime.of(8, 20, 30));//时分秒
System.out.println(LocalTime.of(8, 20, 30, 150));//时分秒纳秒
LocalTime mTime = LocalTime.of(8, 20, 30, 150);

//is系列的方法
System.out.println(nowTime.isBefore(mTime));
System.out.println(nowTime.isAfter(mTime));

//with系列的方法，只能修改时、分、秒
System.out.println(nowTime.withHour(10));

//plus系列的方法，只能修改时、分、秒
System.out.println(nowTime.plusHours(10));
```

LocalDateTime 与 LocalDate 和 LocalTime的相互的转换：

```
 LocalDateTime today = LocalDateTime.now();
 System.out.println(today);
 System.out.println(today.toLocalDate());
 System.out.println(today.toLocalTime());
```

Duration 时间间隔（秒，纳，秒）

```
// 本地日期时间对象。
LocalDateTime today = LocalDateTime.now();
System.out.println(today);

// 出生的日期时间对象
LocalDateTime birthDate = LocalDateTime.of(2000, 1, 1, 0, 0, 0);
System.out.println(birthDate);

Duration duration = Duration.between(birthDate, today);//第二个参数减第一个参数
System.out.println("相差的时间间隔对象:" + duration);

System.out.println("===========================================");
System.out.println(duration.toDays());//两个时间差的天数
System.out.println(duration.toHours());//两个时间差的小时数
System.out.println(duration.toMinutes());//两个时间差的分钟数
System.out.println(duration.toMillis());//两个时间差的毫秒数
System.out.println(duration.toNanos());//两个时间差的纳秒数
```

Period 时间间隔（年，月，日）

```
// 当前本地 年月日
LocalDate today = LocalDate.now();
System.out.println(today);

// 生日的 年月日
LocalDate birthDate = LocalDate.of(2000, 1, 1);
System.out.println(birthDate);

Period period = Period.between(birthDate, today);//第二个参数减第一个参数

System.out.println("相差的时间间隔对象:" + period);
System.out.println(period.getYears());
System.out.println(period.getMonths());
System.out.println(period.getDays());

System.out.println(period.toTotalMonths());
```

ChronoUnit 时间间隔（所有单位）

```
// 当前时间
LocalDateTime today = LocalDateTime.now();
System.out.println(today);
// 生日时间
LocalDateTime birthDate = LocalDateTime.of(2000, 1, 1,0, 0, 0);
System.out.println(birthDate);

System.out.println("相差的年数:" + ChronoUnit.YEARS.between(birthDate, today));
System.out.println("相差的月数:" + ChronoUnit.MONTHS.between(birthDate, today));
System.out.println("相差的周数:" + ChronoUnit.WEEKS.between(birthDate, today));
System.out.println("相差的天数:" + ChronoUnit.DAYS.between(birthDate, today));
System.out.println("相差的时数:" + ChronoUnit.HOURS.between(birthDate, today));
System.out.println("相差的分数:" + ChronoUnit.MINUTES.between(birthDate, today));
System.out.println("相差的秒数:" + ChronoUnit.SECONDS.between(birthDate, today));
System.out.println("相差的毫秒数:" + ChronoUnit.MILLIS.between(birthDate, today));
System.out.println("相差的微秒数:" + ChronoUnit.MICROS.between(birthDate, today));
System.out.println("相差的纳秒数:" + ChronoUnit.NANOS.between(birthDate, today));
System.out.println("相差的半天数:" + ChronoUnit.HALF_DAYS.between(birthDate, today));
System.out.println("相差的十年数:" + ChronoUnit.DECADES.between(birthDate, today));
System.out.println("相差的世纪(百年)数:" + ChronoUnit.CENTURIES.between(birthDate,
today));
System.out.println("相差的千年数:" + ChronoUnit.MILLENNIA.between(birthDate, today));
System.out.println("相差的纪元数:" + ChronoUnit.ERAS.between(birthDate, today));
```

包装类：基本类型对应的对象

Java提供了两个类型系统，基本类型与引用类型，使用基本类型在于效率，然而很多情况，会创建对象使用，因为对象可以做更多的功能，如果想要我们的基本类型像对象一样操作，就可以使用基本类型对应的包装类，如下：

基本类型	对应的包装类（位于java.lang包中）
byte	Byte
short	Short
int	**Integer**
long	Long
float	Float
double	Double
char	**Character**
boolean	Boolean

Integer类

public Integer(int value)	根据 int 值创建 Integer 对象(过时)
public Integer(String s)	根据 String 值创建 Integer 对象(过时)
public static Integer valueOf(int i)	返回表示指定的 int 值的 Integer 实例
public static Integer valueOf(String s)	返回保存指定String值的 Integer 对象
static string tobinarystring(int i)	得到二进制
static string tooctalstring(int i)	得到八进制
static string toHexstring(int i)	得到十六进制
static int parseInt(string s)	将字符串类型的整数转成int类型的整数

自动装箱和自动拆箱：

JDK5后对数据包装类的邢增的特性：自动装箱和自动拆箱

```
Integer i = 4;//自动装箱。相当于Integer i = Integer.valueOf(4);
i = i + 5;//等号右边：将i对象转成基本数值(自动拆箱) i.intValue() + 5;
//加法运算完成后，再次装箱，把基本数值转成对象。
```

获取包装类的对象我们只需要进行赋值即可不需要我们的调用方法

常用的方法：

```
/*
            public static string tobinarystring(int i) 得到二进制
            public static string tooctalstring(int i) 得到八进制
            public static string toHexstring(int i) 得到十六进制
            public static int parseInt(string s) 将字符串类型的整数转成int类型的整数
 */
```

底层原理

建议：获取Integer对象的时候不要自己new，而是采取直接赋值或者静态方法valueOf的方式

因为在实际开发中，-128~127之间的数据，用的比较多。如果每次使用都是new对象，那么太浪费内存了。

所以，提前把这个范围之内的每一个数据都创建好对象，如果要用到了不会创建新的，而是返回已经创建好的对象。

```
//1.利用构造方法获取Integer的对象(JDK5以前的方式)
/*Integer i1 = new Integer(1);
        Integer i2 = new Integer("1");
        System.out.println(i1);
        System.out.println(i2);*/

//2.利用静态方法获取Integer的对象(JDK5以前的方式)
Integer i3 = Integer.valueOf(123);
Integer i4 = Integer.valueOf("123");
Integer i5 = Integer.valueOf("123", 8);

System.out.println(i3);
System.out.println(i4);
System.out.println(i5);

//3.这两种方式获取对象的区别(掌握)
//底层原理：
//因为在实际开发中，-128~127之间的数据，用的比较多。
//如果每次使用都是new对象，那么太浪费内存了
//所以，提前把这个范围之内的每一个数据都创建好对象
//如果要用到了不会创建新的，而是返回已经创建好的对象。
Integer i6 = Integer.valueOf(127);
Integer i7 = Integer.valueOf(127);
System.out.println(i6 == i7);//true

Integer i8 = Integer.valueOf(128);
Integer i9 = Integer.valueOf(128);
System.out.println(i8 == i9);//false

//因为看到了new关键字，在Java中，每一次new都是创建了一个新的对象
//所以下面的两个对象都是new出来，地址值不一样。
/*Integer i10 = new Integer(127);
        Integer i11 = new Integer(127);
```

```
        System.out.println(i10 == i11);

        Integer i12 = new Integer(128);
        Integer i13 = new Integer(128);
        System.out.println(i12 == i13);*/
```

常见的算法：

查找算法：

基本查找：

顺序查找也称为线形查找，属于无序查找算法。从数据结构线的一端开始，顺序扫描，依次将遍历到的结点与要查找的值相比较，若相等则表示查找成功；若遍历结束仍没有找到相同的，表示查找失败。

```
public class A01_BasicSearchDemo1 {
    public static void main(String[] args) {
        //基本查找/顺序查找
        //核心：
        //从0索引开始挨个往后查找

        //需求：定义一个方法利用基本查找，查询某个元素是否存在
        //数据如下：{131, 127, 147, 81, 103, 23, 7, 79}


        int[] arr = {131, 127, 147, 81, 103, 23, 7, 79};
        int number = 82;
        System.out.println(basicSearch(arr, number));

    }

    //参数：
    //一：数组
    //二：要查找的元素

    //返回值：
    //元素是否存在
    public static boolean basicSearch(int[] arr, int number){
        //利用基本查找来查找number在数组中是否存在
        for (int i = 0; i < arr.length; i++) {
            if(arr[i] == number){
                return true;
            }
        }
        return false;
    }
}
```

二分查找

也称为是折半查找，属于有序查找算法。用给定值先与***中间结点***比较。比较完之后有三种情况：

- 相等

  说明找到了

- 要查找的数据比中间节点小

  说明要查找的数字在中间节点左边

- 要查找的数据比中间节点大

  说明要查找的数字在中间节点右边

```
public static int halfSearch(int[] arr, int number){
      int min=0;
      int max=arr.length-1;
      while (true){
          if(min>max){
              return -1;
          }

          int mid=(max+min)/2;

          if(number<arr[mid]){
             max=mid-1;
          }
          else if(number>arr[mid]){
              min=mid+1;
          }else {
              return mid;
          }
      }
```

插值查找

基于二分查找算法，将查找点的选择改进为自适应选择，可以提高查找效率。当然，差值查找也属于有序查找。

```
    public static int inserthalfSearch(int[] arr, int number){
        int min=0;
        int max=arr.length-1;
        while (true){
            if(min>max){
                return -1;
            }

            int mid=min+(number-arr[min])/(arr[max]-arr[min])*(max-min);

            if(number<arr[mid]){
                max=mid-1;
            }
            else if(number>arr[mid]){
```

```
                min=mid+1;
            }else {
                return mid;
            }
        }
    }
```

斐波那契查找

Arrays:

array的方法基本上都是使用的静态修饰的，因此我们可以不需要创建对象，直接使用就可以。

```
static String toString(boolean[] a)
          返回指定数组内容的字符串表示形式。
```

数组的字符串化：

```
package com.itheima.test11;

import java.util.Arrays;

public class Arraystest_demo {
    public static void main(String[] args) {
        int[] arr={1,2,3,4,5,6,7,8,9};
        System.out.println(Arrays.toString(arr));
    }
}
数组转成字符串输出
```

数组的排序

默认下对基本的数据进行排序

```
static void sort(int[] a)
          对指定的 int 型数组按数字升序进行排序。
```

```
package com.itheima.test11;

import java.util.Arrays;

public class Arraystest_demo {
    public static void main(String[] args) {
        int[] arr={1,3,2,10,9,6,7,5,4};
        System.out.println(Arrays.toString(arr));
        Arrays.sort(arr);
        System.out.println(Arrays.toString(arr));
    }
}
```

自定义规则的数据的排序：要求必须为引用数据类型

如果数据为基本的数据类型需要将其转换为引用数据类型。

sort的底层的原理，插入+二分查找的方式进行排序

默认认为我们的0索引为有序的序列，序列到最后默认认为是无需的序列

遍历无序的序列，得到里面的每一个的元素，假设当前遍历到的是元素A

吧A望有序的序列中进行插入，在插入的时候，利用二分查找确定对应的插入点。

比较的规则就是我们comparator的方法体

O1为我们的无序的数组中的遍历元素

O2为我们的有序序列中的元素

返回值：

负数：表示当前需要插入的元素是小的放在前面

正数：表示当前要插入的元素是大的放在后面

0：表示当前的元素和比较的元素是一样的放在后面

```
package com.itheima.test11;

import java.util.Arrays;
import java.util.Calendar;
import java.util.Comparator;

public class Arraystest_demo {
    public static void main(String[] args) {
        Integer[] arr={1,3,2,10,9,6,7,5,4};
        System.out.println(Arrays.toString(arr));
        Arrays.sort(arr, 0, 9, new Comparator<Integer>() {
                    @Override
                    public int compare(Integer o1, Integer o2) {
                        System.out.println("_____________________");
                        System.out.println("o1:"+o1);
```

```
                        System.out.println("o2:"+o2);
                        return o1-o2;
                    }
                });
                //第二个参数是一个接口，我们需要传递这个接口的实现类对象作为排序的规则
                System.out.println(Arrays.binarySearch(arr, 4));
        System.out.println(Arrays.toString(arr));
    }
}
```

```
_____________________首先其默认0索引为有序，之后的默认为无序的
o1:3
o2:1
_____________________遍历无序，得到3和1比较，3-1为正默认其为大，因此放后面
o1:2
o2:3
_____________________现在的有序的数列为1，3 遍历无序的为2，2 和3比较，得负数，因此为小放前面
o1:2
o2:3
_____________________然后再吧2和1 做比较得到1，为正，为大放后面
o1:2
o2:1
_____________________
o1:10
o2:2
_____________________
o1:10
o2:3
_____________________升序：
```

降序

```
package com.itheima.test11;

import java.util.Arrays;
import java.util.Calendar;
import java.util.Comparator;

public class Arraystest_demo {
    public static void main(String[] args) {
        Integer[] arr={1,3,2,10,9,6,7,5,4};
        System.out.println(Arrays.toString(arr));
        Arrays.sort(arr, 0, 9, new Comparator<Integer>() {
                    @Override
                    public int compare(Integer o1, Integer o2) {
                        System.out.println("_____________________");
                        System.out.println("o1:"+o1);
                        System.out.println("o2:"+o2);
                        return o2-o1;
                    }
```

```
                });
                //第二个参数是一个接口，我们需要传递这个接口的实现类对象作为排序的规则
                System.out.println(Arrays.binarySearch(arr, 4));
        System.out.println(Arrays.toString(arr));
    }
}
```

数组的填充：

```
package com.itheima.test11;

import java.util.Arrays;

public class Arraystest_demo1 {
    public static void main(String[] args) {
        int Arr[]=new int[27];
        Arrays.fill(Arr,100);
        System.out.println(Arrays.toString(Arr));
    }
}
```

数组的比较

```
package com.itheima.test11;

import java.util.Arrays;

public class Arraystest_demo1 {
    public static void main(String[] args) {
        int Arr[]=new int[27];
        int Arr1[]=new int[27];
        Arrays.fill(Arr,100);
        Arrays.fill(Arr1,100);
        System.out.println(Arrays.equals(Arr, Arr1));

    }
}
```

数组的复制

```
package com.itheima.test11;
import java.util.Arrays;
public class Arraystest_demo1 {
    public static void main(String[] args) {
        int Arr[]=new int[27];
        int Arr1[]=new int[27];
        Arrays.fill(Arr,100);
        Arrays.fill(Arr1,100);
        System.out.println(Arrays.equals(Arr, Arr1));
```

```
        int[] ints = Arrays.copyOf(Arr, 28);
        System.out.println(Arrays.toString(ints));
    }
}
```

二分查找

二分查找的元素必须是升序的、

二分法查找元素的时候，元素首先必须有序且必须为升序

如果元素是存在的，那返回的是真实的索引，如果元素是不存在的那么返回的是:-插入点-1

出现-插入点-1的原因，假如我们的出现的插入点为0 的时候负的插入点就会还是0 这个就会导致粗错

例如错误案例：

```
package com.itheima.test11;

import java.util.Arrays;
import java.util.Calendar;
import java.util.Comparator;

public class Arraystest_demo {
    public static void main(String[] args) {
        int[] arr={1,3,2,10,9,6,7,5,4};
        System.out.println(Arrays.toString(arr));
        //Arrays.sort(arr,0,9);
        System.out.println(Arrays.binarySearch(arr, 4));
        System.out.println(Arrays.toString(arr));
    }

}
```

正确的案例：必须先排序后进行二分查找

```
package com.itheima.test11;

import java.util.Arrays;
import java.util.Calendar;
import java.util.Comparator;

public class Arraystest_demo {
    public static void main(String[] args) {
        int[] arr={1,3,2,10,9,6,7,5,4};
        System.out.println(Arrays.toString(arr));
        Arrays.sort(arr,0,9);
        System.out.println(Arrays.binarySearch(arr, 4));
```

lamada表达式：

简化我们的代码

函数式的编程：强调做什么，而不是谁来做

需要我们的接口中有且只有一个方法

可以采用我们的lamada的表达式来简写

标准的格式：

```
（方法的形参）->{
方法体
}
```

lamada表达式用来间写匿名内部类的书写

只能简化函数式接口的匿名内部类的写法

什么叫做函数式的接口：

有且只有一个抽象方法的接口叫做函数式的接口

```
package com.itheima.test11;

import java.util.Arrays;
import java.util.Comparator;

public class lamada_demo1 {
    public static void main(String[] args) {
        Integer[] arr={1,3,2,10,9,6,7,5,4};
        System.out.println(Arrays.toString(arr));
        Arrays.sort(arr, 0, 9, (Integer o1, Integer o2)->o2-o1);
        //第二个参数是一个接口，我们需要传递这个接口的实现类对象作为排序的规则
        System.out.println(Arrays.binarySearch(arr, 4));
        System.out.println(Arrays.toString(arr));
    }
    }
```

举例：

```
package com.itheima.test11;

import java.util.Arrays;
import java.util.Comparator;

public class lamada_demo1 {
    public static void main(String[] args) {
        method(()-> System.out.println("hello")
        );
    }
    public static void method(Swim s){
```

```
        s.swimming();
    }
    interface Swim{
        public abstract void swimming();
    }
    }
```

lamada表达式的省略的形式：

可以推导可以省略的核心思想

```
public static void main(String[] args) {
        Integer[] arr={1,3,2,10,9,6,7,5,4};
        System.out.println(Arrays.toString(arr));
        Arrays.sort(arr, 0, 9, (o1,o2)->o2-o1);
        //第二个参数是一个接口，我们需要传递这个接口的实现类对象作为排序的规则
        System.out.println(Arrays.binarySearch(arr, 4));
        System.out.println(Arrays.toString(arr));


    }
```

练习：

```
package com.itheima.test11;
import java.util.Arrays;
public class lamada_demo2 {
    public static void main(String[] args) {
        String[] arr={"AAA","BBBB","CCCC","SSSSSS"};
        System.out.println(Arrays.toString(arr));
        Arrays.sort(arr, 0, 4, (o1,o2)->o2.length()-o1.length());
        //第二个参数是一个接口，我们需要传递这个接口的实现类对象作为排序的规则
       System.out.println(Arrays.toString(arr));
    }
    }
```

通过长度比较字符串并进行排序的

```
package com.itheima.test11;

import java.util.Arrays;
import java.util.Comparator;

public class test {
    public static void main(String[] args) {
        Boyfriends b1=new Boyfriends("zhangsan",18,178);
        Boyfriends b2=new Boyfriends("wangsa",19,180);
        Boyfriends b3=new Boyfriends("zhoasanze",20,167);
        Boyfriends[] boy={b1,b2,b3};
        sort(boy);
        System.out.println(Arrays.toString(boy));
```

```
    }
     public static Boyfriends[] sort(Boyfriends[] boy) {
         Arrays.sort(boy, (o1, o2) -> {
         double temp = o2.getAge()-o1.getAge();
         temp= temp== 0 ? o2.getHeight()-o1.getHeight() : temp;
         temp= temp== 0 ? o2.getName().length()-o1.getName().length():temp;
         if(temp>0){
             return 1;
         } else if (temp<=0) {
             return -1;
         }else
             return 0;
         });
         return boy;
     }
}
```

不死神兔：

第一种遍历数组赋值

```
        int[] arr=new int[12];
        arr[0]=1;
        arr[1]=1;
        for (int i = 2; i < arr.length; i++) {
            arr[i]=arr[i-1]+arr[i-2];
        }
        System.out.println(arr[11]);
```

第二种采用递归的方法

```
    public static int getsum(int month){
        if (month==1||month==2)
            return 1;
        else
            return getsum(month-1)+getsum(month-2);
    }
}
```

猴子吃桃：经典递归

每天吃一半的基础上多吃一个

```
        int sum=getsum(1);
        System.out.println(sum);
    public static int getsum(int month){
        if (month<=0 || month>=11){
            System.out.println("error");
            return -1;}
        if (month==10)
            return 1;
        return (getsum(month+1)+1)*2;
    }
```

day	num	计算
10	1	1
9	4	(1+1) *2
8	10	(4+1) *2

正着推理:

day	num
8	10
9	4
10	1

依赖

$$Num[today] = (Num[nextday] + 1) * 2$$

小明爬楼梯

小明爬楼梯的方法，首先有时候爬一层有时候爬俩层，如果一共有二十层怎么爬？有多少种方法?

分析:

层数	num	算法
1	1	1
2	2	2
3	3	1+2
4	5	2+3

```
package com.itheima.test11;
```

```
public class test5 {
    public static void main(String[] args) {
        System.out.println(getsum(7));
    }
    public static int getsum(int layzer){
       if(layzer==1){
           return 1;
       }else if (layzer==2){
           return 2;
       }
       else
           return getsum(layzer-1)+getsum(layzer-2);
    }
}
```

依赖：

$$getsum(layzer) = getsum(layzer-1) + getsum(layzer-2);$$

集合进阶：

集合的体系结构：

collection：单列的集合

collection是单列集合的祖宗接口，他的功能是全部的单列集合都可以继承使用的

collection的通用的方法：

```
boolean add(E e)
          确保此 collection 包含指定的元素（可选操作）。
boolean addAll(Collection<? extends E> c)
          将指定 collection 中的所有元素都添加到此 collection 中（可选操作）。
void clear()
          移除此 collection 中的所有元素（可选操作）。
boolean contains(Object o)
          如果此 collection 包含指定的元素，则返回 true。
boolean containsAll(Collection<?> c)
          如果此 collection 包含指定 collection 中的所有元素，则返回 true。
boolean equals(Object o)
          比较此 collection 与指定对象是否相等。
int hashCode()
          返回此 collection 的哈希码值。
boolean isEmpty()
          如果此 collection 不包含元素，则返回 true。
Iterator<E> iterator()
          返回在此 collection 的元素上进行迭代的迭代器。
boolean remove(Object o)
          从此 collection 中移除指定元素的单个实例，如果存在的话（可选操作）。
boolean removeAll(Collection<?> c)
          移除此 collection 中那些也包含在指定 collection 中的所有元素（可选操作）。
```

```
 boolean retainAll(Collection<?> c)
          仅保留此 collection 中那些也包含在指定 collection 的元素（可选操作）。
 int size()
          返回此 collection 中的元素数。
 Object[] toArray()
          返回包含此 collection 中所有元素的数组。
<T> T[]
 toArray(T[] a)
          返回包含此 collection 中所有元素的数组；返回数组的运行时类型与指定数组的运行时类型相
同。
```

首先Collection是一个接口我们没有办法去创建他的对象只能创建我们的接口的实现类对象。

contain:

contain的底层其实是依赖于我们的equal 的

所以当我们想要去实现类似于学生的对象的检索比较的时候我们需要重写这个equal的方法：

```
public class collectiondemo2 {
    //接口没办法直接创建我们的目标的对象，只能根据我们的实现类来使用
    //采用这种方法来学习我们的collection中的方法
    public static void main(String[] args) {
        Collection<student> stu=new ArrayList<>();
        stu.add(new student("zhangsan",18,"1234",187));
        stu.add(new student("zhangsan1",19,"1234",187));
        stu.add(new student("zhangsan2",20,"1234",187));
        stu.add(new student("zhangsan3",21,"1234",187));
        stu.add(new student("zhangsan",18,"1234",187));
        System.out.println(stu);
        System.out.println(stu.contains(new student("zhangsan", 18, "1234", 187)));
    }
}
```

重写的equal的方法：

```
 @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        student student = (student) o;
        return age == student.age && height == student.height &&
Objects.equals(name, student.name) && Objects.equals(id, student.id);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, id, height);
    }
```

Collection系列的元素的遍历：

三大遍历的方式：迭代器遍历 增强for循环 lambda表达式

迭代器遍历

迭代器遍历我们的集合
Iterator<E> iterator()
返回在此 collection 的元素上进行迭代的迭代器。
注意事项：
迭代器遍历结束之后，指针不会复位
循环中只能用一次next的方法
迭代器遍历的时候不能用集合的方法去增加或者删除（可能会导致迭代器中的游标的位置和真实的数据位置不匹配报错）
如果必须进行删除的操作，可以使用迭代器的方法remove进行删除的操作
迭代器的指针再遍历完之后不会复位，重新遍历的只能重新获取一个新的迭代器的对象。
next的方法获取元素并移动指针

举例：

```
public class collectiondemo2 {
    //接口没办法直接创建我们的目标的对象，只能根据我们的实现类来使用
    //采用这种方法来学习我们的collection中的方法
    public static void main(String[] args) {
        Collection<student> stu=new ArrayList<>();
        stu.add(new student("zhangsan",18,"1234",187));
        stu.add(new student("zhangsan1",19,"1234",187));
        stu.add(new student("zhangsan2",20,"1234",187));
        stu.add(new student("zhangsan3",21,"1234",187));
        stu.add(new student("zhangsan",18,"1234",187));
        System.out.println(stu);
        System.out.println(stu.contains(new student("zhangsan", 18, "1234", 187)));
        Iterator<student> it=stu.iterator();
        //next获取当前的元素的位置，然后下次将迭代器的对象移动到下一个位置
        System.out.println(it.next());
        System.out.println(it.next());
        System.out.println(it.next());
        //hashnext 判断当前的位置是否存在元素，存在元素则返回true ,否则返回false
        System.out.println(it.hasNext());
    }
}
```

迭代器的遍历增删元素的时候：

```
  while (it.hasNext()){
            String str= it.next().getName();
            if("zhangsan".equals(str)){
                stu.remove(new student("zhangsan",18,"1234",187));
                System.out.println("已经删除");
            }
        }
        System.out.println(stu);
    }
```

报出并发异常的错误：ConcurrentModificationException

解决的方法：

```
 while (it.hasNext()){
            String str= it.next().getName();
            if("zhangsan".equals(str)){
                it.remove();使用迭代器中的删除的方法
                System.out.println("已经删除");
            }
        }
        System.out.println(stu);
```

增强for的遍历：

```
所有的数组和单列的集合才可以使用增强for的遍历方法

增强for的底层就是迭代器，为了简化迭代器的操作而实现的
    快速的生成的方式：
    集合名字.for enter 生成
增强for的细节注意点
        就是再=修改增强for的第三方的变量是不影响原本的集合中的数据的
```

```
public class Arraystest_demo2 {
    //接口没办法直接创建我们的目标的对象，只能根据我们的实现类来使用
    //采用这种方法来学习我们的collection中的方法
    public static void main(String[] args) {
        Collection<student> stu=new ArrayList<>();
        stu.add(new student("zhangsan",18,"1234",187));
        stu.add(new student("zhangsan1",19,"1234",187));
        stu.add(new student("zhangsan2",20,"1234",187));
        stu.add(new student("zhangsan3",21,"1234",187));
        stu.add(new student("zhangsan",18,"1234",187));
        System.out.println(stu);
        System.out.println(stu.contains(new student("zhangsan", 18, "1234", 187)));
        for (student s: stu){
            System.out.println(s);
        }
    }
}
```

lamada表达式的遍历：

得益于Jdk 8 lamada技术，提供一种简介的遍历的方法

```
default void forEach(Consumer<? super T> action)
```

集合lamada遍历数组或者集合

方法的底层，其实也就是自己遍历所有的集合，得到相应的元素，并将元素传递给我们的accept的方法，例如我们的重写的是打印这对象元素，其对应的也就是获取到每一个元素并将其打印

采用匿名内部类的方式实现

```
public class collectiondemo3 {
    //接口没办法直接创建我们的目标的对象，只能根据我们的实现类来使用
    //采用这种方法来学习我们的collection中的方法
    public static void main(String[] args) {
        Collection<student> stu=new ArrayList<>();
        stu.add(new student("zhangsan",18,"1234",187));
        stu.add(new student("zhangsan1",19,"1234",187));
        stu.add(new student("zhangsan2",20,"1234",187));
        stu.add(new student("zhangsan3",21,"1234",187));
        stu.add(new student("zhangsan",18,"1234",187));
        System.out.println(stu);
        System.out.println(stu.contains(new student("zhangsan", 18, "1234", 187)));
        //首先使用匿名内部类的方式实现这个
        stu.forEach(new Consumer<student>() {
            @Override
            public void accept(student student) {
                System.out.println(student);
            }
        });
    }
}
```

改成lamada表达式

改成lamada的形式：

```
public class collectiondemo3 {
    //接口没办法直接创建我们的目标的对象，只能根据我们的实现类来使用
    //采用这种方法来学习我们的collection中的方法
    public static void main(String[] args) {
        Collection<student> stu=new ArrayList<>();
        stu.add(new student("zhangsan",18,"1234",187));
        stu.add(new student("zhangsan1",19,"1234",187));
        stu.add(new student("zhangsan2",20,"1234",187));
        stu.add(new student("zhangsan3",21,"1234",187));
        stu.add(new student("zhangsan",18,"1234",187));
        System.out.println(stu);
        System.out.println(stu.contains(new student("zhangsan", 18, "1234", 187)));
        //首先使用匿名内部类的方式实现这个
        stu.forEach((student)->{
                System.out.println(student);
            }
```

```
        );
    }
}
```

list系列:有序可重复有索引

ArrayList:底层原理

```
arraylist 的底层是数组结构的

1，空参构造的时候，arraylist 会创建一个默认的长度为0的数组。
2，添加第一个的元素的时候，底层会创建一个新的长度为10的数组
3，size 有俩层意义，一表示元素的个数 其二表示元素下次存入的位置索引；
4，当数据存满的时候，就会扩充为原来的1.5倍
5如果一次性存入多个元素，1.5倍放不下的时候，则新创建的数组的长度以实际为准。
源码：
public ArrayList() {
     this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }
 this.elementData本质上是个数组；
 DEFAULTCAPACITY_EMPTY_ELEMENTDATA    为长度是零的数组
add的源码：
会先判断size和自定义的数组的长度进行一个比较
如果发现其存储的元素的size已经到达最大的数组的长度，这里会使用grow进行扩容的操作。
通过扩容的操作得到我们的新的数组，然后使用copy到新的数组中；
      参数一：元素的名字
      参数二：集合的数组得名字
      参数三：集合的长度当前的元素存储的位置


private void add(E e, Object[] elementData, int s) {
        if (s == elementData.length)
            elementData = grow();
        elementData[s] = e;
        size = s + 1;
    }
 size = s + 1;其实就是我们的实际的数据的长度其实也是我们位置的索引。


elementData = grow();执行数组的扩容
扩容的方法：传入的参数为最小的扩容容量
private Object[] grow(int minCapacity) {
        int oldCapacity = elementData.length;记录原来的容量
        if (oldCapacity > 0 || elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
            int newCapacity = ArraysSupport.newLength(oldCapacity,
                    minCapacity - oldCapacity, /* minimum growth */
                    oldCapacity >> 1           /* preferred growth */);数组的扩容右移一
位 默认就是扩容原来的一半，也就是原来的1.5倍。
 return elementData =Arrays.copyOf(elementData,newCapacity);
  }
 else {
```

```
 return elementData = new Object[Math.max(DEFAULT_CAPACITY, minCapacity)];第一次的创建的时候会产生一个10 的数组。
        }
    }
```

扩容最底层的代码：

```
    public static int newLength(int oldLength, int minGrowth, int prefGrowth) {
        // preconditions not checked because of inlining
        // assert oldLength >= 0
        // assert minGrowth > 0

        int prefLength = oldLength + Math.max(minGrowth, prefGrowth); // might overflow
        if (0 < prefLength && prefLength <= SOFT_MAX_ARRAY_LENGTH) {
            return prefLength;
        } else {
            // put code cold in a separate method
            return hugeLength(oldLength, minGrowth);
        }
    }
```

`int prefLength = oldLength + Math.max(minGrowth,prefGrowth);` 拿着至少需要增加的容量和默认需要增加的容量做一个比较，选取一个最大的数值、在集合中默认不仅是一次加一，有时候还会增加许多的元素

当我们一下子增加若干的数据的时候，就会出现一`Math.max(minGrowth,prefGrowth)`，`minGrowth` 大于`pregrowth`,因此我们就会开始采用最大的最为扩容的容量的浮动。

LinkedList的底层原理:

`Linklist`是一个双链表的基本的数据结构，查询慢，增删满，但是如果操作的是首尾的元素的时候，速度很快。

链表的节点的源码，

```
private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;

        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }
```

add的代码：

```
public boolean add(E e) {
        linkLast(e);
        return true;
    }
void linkLast(E e) {
```

```
        final Node<E> l = last;
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode;
        if (l == null)
            first = newNode;
        else
            l.next = newNode;
        size++;
        modCount++;
    }
```

这里将我们新创建的节点的地址赋值给我们的last

然后会做一个判断，判断是否我们的节点的Last 是不是null.若果是则first 为新节点的地址

首先第一步：传入的数据的Last肯定是null,这个是空的时候，将新节点传入last,并将头指针指向新节点。其实此时创建的一个节点的头尾指针都为空

接下来的第二部再执行的时候，传入的这个创建一个新的节点，该节点接收到了之前创建的last的地址，二者相连接，再将我们的last指向我们的现在的新节点，并将这个新节点的地址传给前一个节点的地址，这就实现了后面的链接前面的

如果不是null,则将last的地址赋值给我们的新节点，则二者实现链接，接下来将创建的新节点的地址赋值给last

接下来就会判断我们的L是不是空，但是实际上的L此时已经有了地址，那么这时候就会执行else。将第一个新的节点的地址赋值给L.next(),这就是实现了前后的链接

迭代器的底层原理：

```
Iterator<student> it=stu.iterator();
while (it.hasNext()){
            String str= it.next().getName();
            if("zhangsan".equals(str)){
                stu.remove(new student("zhangsan",18,"1234",187));
                System.out.println("已经删除");
            }
        }
        System.out.println(stu);
    }
```

底层代码：

```
   public Iterator<E> iterator() {
        return new Itr();
    }
```

多次调用这个的方法时候

其实就是创建了多个这个迭代器的对象

```
private class Itr implements Iterator<E> {
        int cursor;       // 指向零索引游标，迭代器中的指针
        int lastRet = -1; // 刚刚操作的索引的位置
        int expectedModCount = modCount;//集合变化的次数
```

```
        // prevent creating a synthetic constructor
        Itr() {}//空参构造

        public boolean hasNext() {
            return cursor != size;
        } //判断光标的位置是否超过集集合的最大的size不相等，则返回true,否则返回一个false

        @SuppressWarnings("unchecked")
        public E next() {
            checkForComodification();//判断当前的集合中最新的变化次数和一开始记录的是否相同，
如果相同则表示数据没有发生变化，如果数据不相同的时候，就是数据使用了集合中的方法添加或者删除元素了
            int i = cursor;//记录当前的指针的指向的位置
            if (i >= size)//超过界限的时候报错，没有这个元素异常
                throw new NoSuchElementException();
            Object[] elementData = ArrayList.this.elementData;//获取到底部的集合数组
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            cursor = i + 1;//赋值给游标位置+1
            return (E) elementData[lastRet = i];//返回游标的位置的数据，数组通过这个的索引
来迭代提取。
        }


    public void remove() {
            if (lastRet < 0)
                throw new IllegalStateException();
            checkForComodification();

            try {
                ArrayList.this.remove(lastRet);
                cursor = lastRet;
                lastRet = -1;
                expectedModCount = modCount;
            } catch (IndexOutOfBoundsException ex) {
                throw new ConcurrentModificationException();
            }
        }
```

结论：避免出现并发修改的异常，在使用迭代器和增强for的循环的时候不要使用集合的方法去修改集合中的元素

list所特有的方法：增加了和索引相关的方法

```
void add(String item, int index)
          向滚动列表中索引指示的位置添加指定的项。
E   remove(int index)    删除指定的索引处 的元素，返回被删除的元素
E   set(int index,E element)    设置指定的索引处的元素，返回被修改的元素
E   get(int index)    获取指定的索引处 的元素，返回指定索引处得到的元素
```

通过索引的删除的操作

```java
package com.itheima.test11;

import java.util.ArrayList;
import java.util.List;

public class Listdemo1 {
    public static void main(String[] args) {

        List<Integer> Li=new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            Li.add(i,i);
        };
        System.out.println(Li);
        System.out.println(Li.remove(1));
        //自动装箱
        Integer i=2;
        //手动装箱
        Integer I1=Integer.valueOf(3);
        //通过索引和元素删除
        Li.remove(i);
        Li.remove(I1);
        System.out.println(Li);
        Li.add(1,1);
        System.out.println(Li);
        System.out.println(Li.get(1));
        Li.set(0,10);
        System.out.println(Li);
    }
}
```

输出：

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
1
[0, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
1
```

List集合的遍历的方式：

```
package com.itheima.test11;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;
import java.util.function.Consumer;

public class Listdemo1 {
    public static void main(String[] args) {

        List<Integer> Li=new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            Li.add(i,i);
        };
        Iterator it=Li.iterator();
        while (it.hasNext()){
            System.out.println(it.next());
        }
        System.out.println("-------------------------");
        Li.forEach(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) {
                System.out.println(integer);
            }
        });
        System.out.println("_____________________________");
        Li.forEach((integer)-> System.out.println(integer));
        System.out.println("_____________________________________");
        for (int i = 0; i < Li.size(); i++) {
            System.out.println(Li.get(i));
        }
        //列表迭代器
        System.out.println("---------------列表迭代器----------------");
        ListIterator si=Li.listIterator();
        while (si.hasNext()){
            Object x=si.next();
            System.out.println(x);
            Integer z=5;
            if (x.equals(z)){
                Integer xs=19;
                si.set(xs);
            }
            System.out.println("返回的索引----------------------");
            System.out.println(si.nextIndex());
        }
        System.out.println(Li);
        System.out.println("--------------倒着迭代的----------------");
        while (si.hasPrevious()){
            System.out.println(si.previous());
```

```
            System.out.println("返回的索引----------------------");
            System.out.println(si.previousIndex());
        }

    }
}
```

Set系列：无序不重复无索引

```
public class demo1 {
    public static void main(String[] args) {

//        创建一个set 的集合
        HashSet<String> sh=new HashSet<>();
        sh.add("gello");
        sh.add("what");
        sh.add("kello");
        System.out.println(sh);
//        不能出现重复的数据，出现重复的数据的时候就无法存入
        System.out.println(sh.add("gello"));
        //增强for循环
        for (String s : sh) {
            System.out.println(s);
        }
        System.out.println("__________________");
        //迭代器
        Iterator i=sh.iterator();
        while (i.hasNext()) {
            System.out.println(i.next());
        }
        System.out.println("__________________");
        //匿名内部类
        sh.forEach(new Consumer<String>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }

        });
        System.out.println("__________________");
        //lamada的方法实现
        sh.forEach((s)-> {
                System.out.println(s);
            }
        );
    }
}
```

Hsahset:

底层为哈希表，属于set的系列的一员

Hsahset->LinkedhashSet:

链表 属于哈希set 系列

Treeset:

底层为树结构，属于set 的一员

泛型的深入：

泛型只能支持引用的数据类型，可以再编译的阶段约束操作的数据类型，并进行检查的。

泛型的格式：<数据类型>

泛型用来约束数据的类型

看门的

注意：Java中的泛型是伪泛型的，其实就是检查一下数据

但是再获取数据的时候就会多做一步，将object 转为对应的泛型的约束的类型

在编译的时候会报错，但是生成的class的中没有泛型

泛型的出现就是为了统一集合中的元素，避免一些问题

确认泛型的输入的时候我们就可以传入该类型及其子类类型

```
public class Genericsdemo1 {
    public static void main(String[] args) {
        ArrayList ls=new ArrayList<>();
        ls.add(123);
        ls.add("123");
        ls.add(new student("zhangsan",12,"123",176));
        System.out.println(ls);
        Iterator iterator = ls.iterator();

        //如果我们默认的数据没有给集合一个给定的数据的类型的话，就将会导致再获取数据的时候得到的就都是object的类型
        //无法获取到他的特有的方法；
        //因此Java提出了泛型，便于后面的操作
        while(iterator.hasNext()){
            Object obj=iterator.next();
            //多态的弊端是不能访问子类的特有的功能
            //因此无法调用我们子类的功能
            System.out.println(obj);
        }
```

```
    }
}
```

泛型可以在很多的地方定义：

```
泛型类：当我们的某一类中的变量的数据类型不确定的时候我们就可以自定义带有泛型的类。
格式：
    修饰符 class 类名<E>{

    }
举例子：
    public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable{

    }
```

列子：自己定义的泛型的类

```
package com.itheima.test11;
import java.util.Arrays;
import java.util.Objects;
/*
自定义的泛型类
 */
public class myarraylist<E>{
    Object[] obj=new Object[10];
    int size;
    public boolean add(E e){
    obj[size]=e;
    size++;
    return true;
    }
    public E get(int index){
        return (E) obj[index];
    }

    @Override
    public String toString() {
        return Arrays.toString(obj);
    }
```

```
泛型的方法：
当我们的方法中的形参的类型不确定的时候我们选择使用俩种方法
1，使用类名后面的定义的泛型
2，或者在方法的申明上定义自己的泛型

格式：
修饰符 <类型> 返回类型 函数名字(类型名 变量名){

}
```

```
public class myarraylist{
    Object[] obj=new Object[10];
    int size;
    public <E> boolean add(E e){
    obj[size]=e;
    size++;
    return true;
    }
```

举例子：泛型的方法其实是在定义的类中我们只有在一个方法中存在类型不确定的问题，那么就可以采用泛型方法来实现，不需要将该类别整成一个泛型的类

比如下面的代码所示：

```
public class listutil {
    private listutil(){

    }
    /*
    参数一就是集合
    参数二就是我们需要添加的元素

     */

    public static<E> void addAll(ArrayList<E> list,E e1,E e2,E e3,E e4){
        list.add(e1);
        list.add(e2);
        list.add(e3);
        list.add(e4);
    }

}
```

不定长的传入：

```
   public static<E> void addAll2(ArrayList<E> list,E...e){
        for (E e1 : e) {
            list.add(e1);
        }
    }
```

泛型接口：

```
修饰符 interface 接口名<类型>{

}
举例：
 public interface list<E>{

}
关键我们怎么使用泛型的接口
    方法一：实现类黑出具体的类型
    public class myarraylist2 implements List<String>{}

    方法二：实现类延续泛型，创建对象的时候再去确定
```

泛型的继承和通配符

```
 泛型不具备继承性，但是数据具有继承性；
     泛型中写的什么类型，那么就只能传递什么类型的shuju
     弊端：
     利用泛型方法有一个小的弊端，此时他可以接受任意的数据类型，它可以接受所有的任意类型的数据
     有的时候我们希望本方法可以传递我们的ye fu zi 的三个类型
     此时我们就可以使用泛型的通配符
     ？表示不确定的类型
     它可以进行一些类型的限定
     ？ extends E:表示可以传递E或者E的一些子类的类型
     ？ super E:表示传递E或者E的所有的父类的类型
```

利用泛型的通配符的使用：

使用的场景：我们不知道类型但是我们知道其可以继承哪一个类的继承体系中的。

```
public class mian_test {
    public static void main(String[] args) {

            ArrayList<ye> list1=new ArrayList<>();
            ArrayList<fu> list2=new ArrayList<>();
            ArrayList<zi> list3=new ArrayList<>();
            ArrayList<students2> list4=new ArrayList<>();
            methhod(list1);
            methhod(list2);
            methhod(list3);
            methhod(list4);
    }
    public static void methhod(ArrayList<?extends ye> list){

    }

}
class  ye{}
class  fu extends ye{}
```

```
class  zi extends fu{}
class  students2{}
```

```
public class mian_test {

    public static void main(String[] args) {

            ArrayList<ye> list1=new ArrayList<>();
            ArrayList<fu> list2=new ArrayList<>();
            ArrayList<zi> list3=new ArrayList<>();
            ArrayList<students2> list4=new ArrayList<>();
            methhod(list1);
            methhod(list2);
            methhod(list3);
            methhod(list4);
    }
    public static void methhod(ArrayList<?super zi> list){

    }

}
class  ye{}
class  fu extends ye{}
class  zi extends fu{}
class  students2{}
```

泛型的练习：

```
package com.itheima.test12;

import java.util.ArrayList;

public class test {
    public static void main(String[] args) {
     ArrayList<cat> ani=new ArrayList<>();
     taididog taidi=new taididog("xiaosan",2);
     hashiqidog hashiqi=new hashiqidog("xiaoha",3);
     bosicat bosi=new bosicat("bosi",1);
     lihuacat lihua=new lihuacat("lihua",3);
//     ani.add(taidi);
     ani.add(bosi);
//     ani.add(hashiqi);
     ani.add(lihua);
     keepPet1(ani);

    }
    public static void keepPet(ArrayList<?extends animal> list){
        for (animal animal : list) {
            animal.eat();
        }
```

```
    }
    public static void keepPet1(ArrayList<?extends cat> list){
        for (cat animal : list) {
            animal.eat();
        }
    }

}
```

Map：多列的集合

双列集合一次需要存储一对数据，分别为键和值

键是不能重复的，数值是可以重复的

键和值是一一对应的，每一个键只能找到自己对应的数值

键加值的这个整体我们称之为”键值对“或者键值对对象，Java中叫做Entry 对象

MAP中常见的API

public interface Map<K,V>将键映射到值的对象。一个映射不能包含重复的键；每个键最多只能映射到一个值。

```
void clear()
          从此映射中移除所有映射关系（可选操作）。
boolean containsKey(Object key)
          如果此映射包含指定键的映射关系，则返回 true。
boolean containsValue(Object value)
          如果此映射将一个或多个键映射到指定值，则返回 true。
Set<Map.Entry<K,V>> entrySet()
          返回此映射中包含的映射关系的 Set 视图。
boolean equals(Object o)
          比较指定的对象与此映射是否相等。
V get(Object key)
          返回指定键所映射的值；如果此映射不包含该键的映射关系，则返回 null。
int hashCode()
          返回此映射的哈希码值。
boolean isEmpty()
          如果此映射未包含键-值映射关系，则返回 true。
Set<K> keySet()
          返回此映射中包含的键的 Set 视图。
V put(K key, V value)
          将指定的值与此映射中的指定键关联（可选操作）。
void putAll(Map<? extends K,? extends V> m)
          从指定映射中将所有映射关系复制到此映射中（可选操作）。
V remove(Object key)
          如果存在一个键的映射关系，则将其从此映射中移除（可选操作）。
int size()
          返回此映射中的键-值映射关系数。
```

```
Collection<V> values()
        返回此映射中包含的值的 Collection 视图。
```

示例:

```
V put(K key, V value)
        将指定的值与此映射中的指定键关联（可选操作）。
    键值不存在为添加的操作： 添加数据的时候，如果键值不存在的时候，直接添加键值对对象到map的集合当中，此时的方法返回的为null
    键值存在的时候为覆盖的操作： 添加数据的时候，如果键值存在的时候，会把原来的键值的对象覆盖并返回被覆盖的对象
```

```
public class demo2 {
    public static void main(String[] args) {

        Map<String, String> map = new HashMap<>();
//        添加元素
        map.put("郭靖","黄蓉");
        map.put("韦小宝","木健斌");
        map.put("应制品","小农女");
        System.out.println(map);
        String value=map.put("韦小宝","霜儿");
        System.out.println(value);
        System.out.println(map);
    }
}
{韦小宝=木健斌, 应制品=小农女, 郭靖=黄蓉}
木健斌
{韦小宝=霜儿, 应制品=小农女, 郭靖=黄蓉}
```

```
V remove(Object key)
        如果存在一个键的映射关系，则将其从此映射中移除（可选操作）。
```

```
        map.remove("韦小宝");
        System.out.println(map);
        输出:
{韦小宝=霜儿, 应制品=小农女, 郭靖=黄蓉}
{应制品=小农女, 郭靖=黄蓉}
```

```
map.clear();
System.out.println(map);
直接清空的操作
```

```
        System.out.println(map.containsKey("韦小宝"));
        System.out.println(map.containsValue("小农女"));
        System.out.println(map.isEmpty());
        System.out.println(map.size());
        System.out.println(map.values());
输出:
true
true
false
3
[霜儿, 小农女, 黄蓉]
```

map 的集合的遍历的方法:

map 的第一种遍历方法: (键找值)

```
public class demo2 {
    public static void main(String[] args) {

        Map<String, String> map = new HashMap<>();
//        添加元素
        map.put("郭靖", "黄蓉");
        map.put("韦小宝", "木健斌");
        map.put("应制品", "小农女");
        Set<String> strings = map.keySet();
        for (String s : strings) {
            System.out.println(map.get(s));
        }
        strings.forEach((String s)-> {
            System.out.println(map.get(s));
            }
        );
        Iterator<String> iterator = strings.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

    }
}
```

map 的第二种遍历方法: (键值对)

```
public class demo2 {
    public static void main(String[] args) {

        Map<String, String> map = new HashMap<>();
//        添加元素,创建的map的对象
        map.put("郭靖", "黄蓉");
        map.put("韦小宝", "木健斌");
        map.put("应制品", "小农女");
//        map的第二种的遍历的方法
```

```
        Set<Map.Entry<String, String>> entries = map.entrySet();
        for (Map.Entry<String, String> entry : entries) {
            String key = entry.getKey();
            String value = entry.getValue();
            System.out.println("键" + key + "值" + value);
        }

        entries.forEach(new Consumer<Map.Entry<String, String>>() {
            @Override
            public void accept(Map.Entry<String, String> stringStringEntry) {
                String key =stringStringEntry.getKey();
                String value = stringStringEntry.getValue();
                System.out.println("键" + key + "值" + value);
            }
        });

        entries.forEach((Map.Entry<String, String> stringStringEntry)-> {
                String key =stringStringEntry.getKey();
                String value = stringStringEntry.getValue();
                System.out.println("键" + key + "值" + value);
            }
        );
        Iterator<Map.Entry<String, String>> iterator = entries.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

    }
}
```

map 的第三种遍历方法：利用lamada的表达式实现这个

```
public class demo2 {
    public static void main(String[] args) {

        Map<String, String> map = new HashMap<>();
//        添加元素,创建的map的对象
        map.put("郭靖", "黄蓉");
        map.put("韦小宝", "木健斌");
        map.put("应制品", "小农女");
//        map的第三种的遍历的方法
        map.forEach(new BiConsumer<String, String>() {
            @Override
            public void accept(String key, String Value) {
                System.out.println(key+"="+ Value);
            }
        });
        map.forEach(( key,Value )->
                System.out.println(key+"="+ Value)
        );
    }
```

```
}
```

foreach 的底层:

```
    default void forEach(BiConsumer<? super K, ? super V> action) {
        Objects.requireNonNull(action);
        for (Map.Entry<K, V> entry : entrySet()) {
            K k;
            V v;
            try {
                k = entry.getKey();
                v = entry.getValue();
            } catch (IllegalStateException ise) {
                // this usually means the entry is no longer in the map.
                throw new ConcurrentModificationException(ise);
            }
            action.accept(k, v);
        }
    }方法的底层还是调用我们的这个Entry,申城我们的entrySet,完成数据的遍历即可
```

Hashmap的学习

属于我们的maP 中的一个实现类

没有额外的实现的学习的方法。直接使用map中得方法即可

Hash 和Hashset的底层原理都是一样的

都是哈希表的结构

```
public class demo3 {
    public static void main(String[] args) {

        HashMap<student, String> map = new HashMap<>();
//          创建三个学生对象
        student st1 = new student("zhangsan", 12);
        student st2 = new student("zhangsan1", 13);
        student st3 = new student("zhangsan2", 14);
        map.put(st1, "甘肃");
        map.put(st2, "兰州");
        map.put(st3, "湖南");
        map.forEach(new BiConsumer<student, String>() {
            @Override
            public void accept(student student, String s) {
                System.out.println(student + s);
            }
        });
    }
    }
```

```
public class demo3 {
    public static void main(String[] args) {

        HashMap<student, String> map = new HashMap<>();
//        创建三个学生对象
        student st1 = new student("zhangsan", 12);
        student st2 = new student("zhangsan1", 13);
        student st3 = new student("zhangsan2", 14);
        student st4 = new student("zhangsan", 12);
        map.put(st1, "甘肃");
        map.put(st2, "兰州");
        map.put(st3, "湖南");
        System.out.println(map.put(st4, "长沙"));
        map.forEach(new BiConsumer<student, String>() {
            @Override
            public void accept(student student, String s) {
                System.out.println(student + s);
            }
        });
    }
    }
```

我们发现其会对比是否是相同的键值，同姓名同年宁的时候就会出现一个覆盖的操作

```
public class demo3 {
    public static void main(String[] args) {

        HashMap<student, String> map = new HashMap<>();
//        创建三个学生对象
        student st1 = new student("zhangsan", 12);
        student st2 = new student("zhangsan1", 13);
        student st3 = new student("zhangsan2", 14);
        student st4 = new student("zhangsan", 12);
        map.put(st1, "甘肃");
        map.put(st2, "兰州");
        map.put(st3, "湖南");
        System.out.println(map.put(st4, "长沙"));
        map.forEach(new BiConsumer<student, String>() {
            @Override
            public void accept(student student, String s) {
                System.out.println(student + s);
            }
        });
        Set<student> students = map.keySet();
        for (student student : students) {
            System.out.println(map.get(student));
        }
        Set<Map.Entry<student, String>> entries = map.entrySet();
        for (Map.Entry<student, String> entry : entries) {
            System.out.println( entry.getKey()+ entry.getValue());
        }
    }
```

```
    }
```

hashset 的遍历，以及其对比键值是否相等的时候，我们需要重写我们的equal和hashcode的方法，完成对应的操作。

小练习：经典的投票的问题，我们可以使用map来完成

```
package com.itheima.test13;

import com.itheima.test4.Animal;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Random;
import java.util.function.BiConsumer;

public class demo4 {
    public static void main(String[] args) {

        HashMap<String,Integer> map=new HashMap<String, Integer>();
        String[] arr={"A","B","C","D"};
        //随机数来模拟八十位学生的投票的行为
        ArrayList<String> list=new ArrayList<>();
        Random rando=new Random();
        for (int i = 0; i < 80; i++) {
            int nextInt = rando.nextInt(arr.length);
            String s = arr[nextInt];
            list.add(s);
        }
//        表示八十个的同学已经完成了投票的行为
        for (String s : list) {
//            判断当前的经景点的名称再我们的MAp中是否是存在的
            if(map.containsKey(s)){
                Integer integer = map.get(s);
                Integer integer1=integer+1;
                map.put(s,integer1);
            }else {
                map.put(s,1);
            }
        }
        map.forEach(new BiConsumer<String, Integer>() {
            @Override
            public void accept(String s, Integer integer) {
                System.out.println("景点的名字为"+s+"景点的投票人数："+integer);
            }
        });
    }
}
```

运行的结果：

景点的名字为A景点的投票人数：24

景点的名字为B景点的投票人数：18

```
景点的名字为C景点的投票人数：19
景点的名字为D景点的投票人数：19
```

```java
public class demo4 {
    public static void main(String[] args) {

        HashMap<String,Integer> map=new HashMap<String, Integer>();
        String[] arr={"A","B","C","D"};
        //随机数来模拟八十位学生的投票的行为
        ArrayList<String> list=new ArrayList<>();
        Random rando=new Random();
        for (int i = 0; i < 80; i++) {
            int nextInt = rando.nextInt(arr.length);
            String s = arr[nextInt];
            list.add(s);
        }
//        表示八十个的同学已经完成了投票的行为
        for (String s : list) {
//            判断当前的经景点的名称再我们的MAp中是否是存在的
            if(map.containsKey(s)){
                Integer integer = map.get(s);
                Integer integer1=integer+1;
                map.put(s,integer1);
            }else {
                map.put(s,1);
            }
        }
        map.forEach(new BiConsumer<String, Integer>() {
            @Override
            public void accept(String s, Integer integer) {
                System.out.println("景点的名字为"+s+"景点的投票人数："+integer);
            }
        });


//        求一个最大的数值
        Integer max=0;
        String index=new String();
        for (String s : map.keySet()) {
           if (map.get(s)>max)
           {
               max= map.get(s);
               index=s;}
        }
        System.out.println(index+max);
    }
}
```

```
得出最多的景点 的投票的人数’
景点的名字为A景点的投票人数：18
景点的名字为B景点的投票人数：29
景点的名字为C景点的投票人数：13
```

```
景点的名字为D景点的投票人数：20
B29
```

Linkedhashmap

```
都有键值所决定的：有序 不重复 无索引
底层的原理的数据结构是哈希表，只是每一个键值对元素有额外多了一个双链表的机制记录存储的顺序的位置
```

```
public class DEMO5 {

    public static void main(String[] args) {
        LinkedHashMap<String,Integer> map=new LinkedHashMap<>();
        map.put("A",1);
        map.put("b",2);
        map.put("c",3);
        map.put("A",4);
        System.out.println(map);
    }
}
```

Treemap

不重复 五索引 可排序

可排序：对键进行排序，

注意可以按照键值的从小到大的顺序完成排序也可以按照自己的规则完成排序

```
代码书写的排序规则

实现comparable的接口，指定比较的规则
compareTo int compareTo(T o)比较此对象与指定对象的顺序。如果该对象小于、等于或大于指定对象，则分别返回负整数、零或正整数。


创建集合的时候传递comparator比较器的对象。指定比较规则
 public class demo6 {
    public static void main(String[] args) {
        TreeMap<Integer,String> map=new TreeMap<>(new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2-o1;
            }
        });
        map.put(1,"奶茶");
        map.put(2,"果汁");
        map.put(3,"面包");
        map.put(8,"COCO");
        map.put(4,"MILK");
        map.put(5,"COLO");
        System.out.println(map);
```

```
    }
}
```

注意我们在定义Treemap的集合的时候，需要在对象中重写我们排序的规则，否则其会报错，以下是俩种我们使用的方法：

使用对象实现comparable的接口完成。

```
package com.itheima.test13;

import java.util.Objects;

public  class student implements Comparable<student>{
    private String name;
    private int age;

    public student() {
    }

    public student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    /**
     * 获取
     * @return name
     */
    public String getName() {
        return name;
    }

    /**
     * 设置
     * @param name
     */
    public void setName(String name) {
        this.name = name;
    }

    /**
     * 获取
     * @return age
     */
    public int getAge() {
        return age;
    }

    /**
     * 设置
     * @param age
```

```
     */
    public void setAge(int age) {
        this.age = age;
    }

    public String toString() {
        return "student{name = " + name + ", age = " + age + "}";
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        student student = (student) o;
        return age == student.age && Objects.equals(name, student.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }

    @Override
    public int compareTo(student o) {
        int i = o.getAge() - this.getAge();
        i = i==0 ? o.getName().compareTo(this.getName()):i;
        return i;
    }
}
```

使用传入 Comparator对象实现

```
public class demo8 {
    public static void main(String[] args) {
        TreeMap<student2,String> map=new TreeMap<>(new Comparator<student2>() {
            @Override
            public int compare(student2 o1, student2 o2) {
                int i = o1.getAge() - o2.getAge();
                i = i==0 ? o1.getName().compareTo(o2.getName()):i;
                return i;
            }
        });
        student2 stu1=new student2("zhangsan",12);
        student2 stu2=new student2("zhangsan1",13);
        student2 stu3=new student2("zhangsan2",14);
        student2 stu4=new student2("zhangsan3",15);
        map.put(stu4,"湖南");
        map.put(stu1, "甘肃");
        map.put(stu2, "兰州");
        map.put(stu3, "湖南");
        System.out.println(map);
```

```
    }
}
```

小练习：统计字符串中的字符的个数

升序输出：

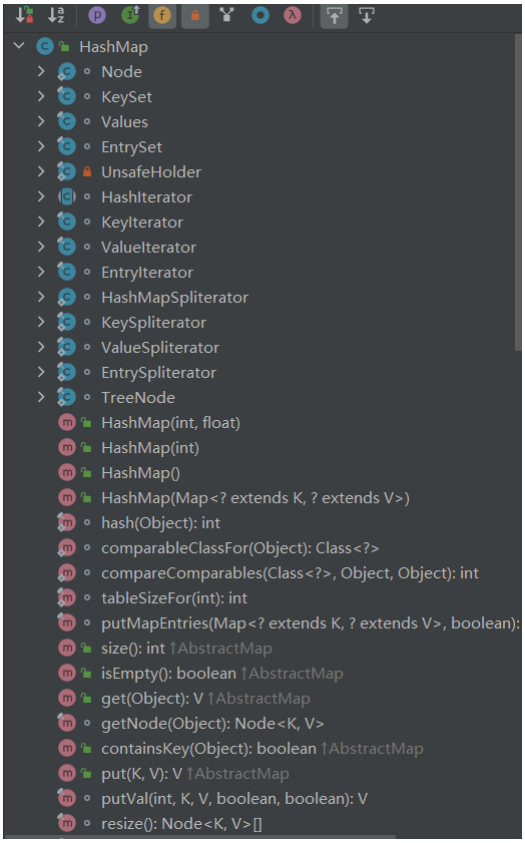
```
public class demo9 {
    public static void main(String[] args) {
        String str="aababcabcdabvde";
        ArrayList<String> list=new ArrayList<>();
        for (int i = 0; i < str.length(); i++) {
            String s= String.valueOf(str.charAt(i));
            System.out.println(s);
            list.add(s);
        }

        TreeMap<String,Integer> map=new TreeMap<>();
        for (String s : list) {
            if (map.containsKey(s)){
                Integer integer = map.get(s);
                integer=integer+1;
                map.put(s,integer);
            }
            else
                map.put(s,1);
        }
        System.out.println(map);
    }
}
```

降序输出：

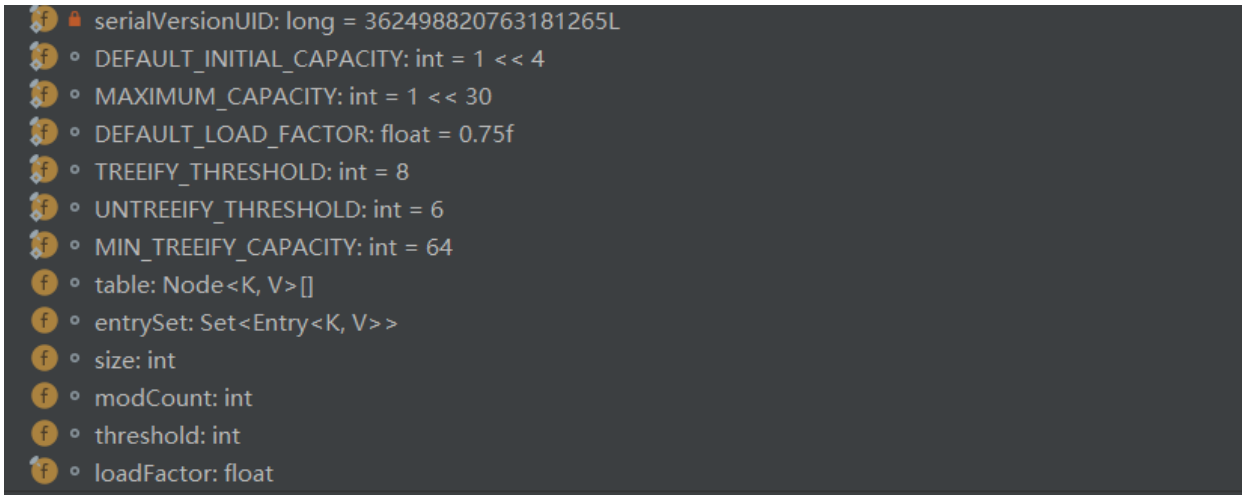
```
public class demo9 {
    public static void main(String[] args) {
        String str="aababcabcdabvde";
        ArrayList<String> list=new ArrayList<>();
        for (int i = 0; i < str.length(); i++) {
            String s= String.valueOf(str.charAt(i));
            System.out.println(s);
            list.add(s);
        }

        TreeMap<String,Integer> map=new TreeMap<>(new Comparator<String>() {
            @Override
            public int compare(String o1, String o2) {
                return o1.compareTo(o2);
            }
        });
```

```
        for (String s : list) {
            if (map.containsKey(s)){
                Integer integer = map.get(s);
                integer=integer+1;
                map.put(s,integer);
            }
            else
                map.put(s,1);
        }
        System.out.println(map);
    }
}
```

HASHMAP的底层源码



哈希属性



hashmap的内部类

hashmap 的内部存储的node

1.1 链表中的键值对对象

包含：

```
int hash;          //键的哈希值
final K key;       //键
V value;           //值
Node<K,V> next;    //下一个节点的地址值
```

1.2 红黑树中的键值对对象

包含：

```
int hash;                //键的哈希值
final K key;             //键
V value;                 //值
TreeNode<K,V> parent;    //父节点的地址值
TreeNode<K,V> left;      //左子节点的地址值
TreeNode<K,V> right;     //右子节点的地址值
boolean red;             //节点的颜色
```

存放的位置：

```
transient Node<K,V>[] table;
其实我们的hashmap的每一个node节点存放的位置；
```

默认的我们的容量：14

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

扩容的容量：

```
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

最大的容量：

```
static final int MAXIMUM_CAPACITY = 1 << 30;
```

构造的方法：

空参构造的时候不创建我们的table

```
2.添加元素
HashMap<String,Integer> hm = new HashMap<>();
//此时的数组table为null
hm.put("aaa" , 111);
hm.put("bbb" , 222);
hm.put("ccc" , 333);
hm.put("ddd" , 444);
hm.put("eee" , 555);
添加元素的时候至少考虑三种情况：
2.1数组位置为null
2.2数组位置不为null，键不重复，挂在下面形成链表或者红黑树
2.3数组位置不为null，键重复，元素覆盖
```

他其实是在我们的put的操作的时候完成的创建的对应的table

```
public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
    }
```

俩个参数

参数一就是我们的键

参数二就是我们的值

返回值：俩种类型

第一种：键值没有重复，返回null

第二种：键值重复的时候。返回的是被覆盖的数值

putVal(hash(key), key, value, false, true)解读

其实就是利用我们的键值计算出对应的哈希的数，载把我们的哈希做一些额外的处理，使得我们得到的hashcode 更加的随机

```
//参数一：键的哈希值
//参数二：键
//参数三：值
//参数四：如果键重复了是否保留
//          true，表示老元素的值保留，不会覆盖
//          false，表示老元素的值不保留，会进行覆盖
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,boolean evict) {
        //定义一个局部变量，用来记录哈希表中数组的地址值。采用局部变量可以使得其效率提高不用每次都去堆里面查找我们的数据
        Node<K,V>[] tab;
        //临时的第三方变量，用来记录键值对对象的地址值
        Node<K,V> p;
        //表示当前数组的长度
        int n;
        //表示索引
         int i;
        //把哈希表中数组的地址值，赋值给局部变量tab
        tab = table;
        if (tab == null || (n = tab.length) == 0){
//1.如果当前是第一次添加数据，底层会创建一个默认长度为16，加载因子为0.75的数组
//2.如果不是第一次添加数据，会看数组中的元素是否达到了扩容的条件
            //如果没有达到扩容条件，底层不会做任何操作
            //如果达到了扩容条件，底层会把数组扩容为原先的两倍，并把数据全部转移到新的哈希表中
            tab = resize();//完成数据的扩容
            //表示把当前数组的长度赋值给n
            n = tab.length;
        }

        //拿着数组的长度跟键的哈希值进行计算
       //计算出当前键值对对象，在数组中应存入的位置
        i = (n - 1) & hash;//index计算出存入的索引
        //获取数组中对应元素的数据
        p = tab[i];


        if (p == null){
```

```
            //底层会创建一个键值对对象，直接放到数组当中
  /*Node<K,V> newNode(int hash, K key, V value, Node<K,V> next) {
        return new Node<>(hash, key, value, next);
        直接创建了一个新的键值对的对象，直接放入到我们数组之中。
    }*/
            tab[i] = newNode(hash, key, value, null);
        }else {
            Node<K,V> e;
            K k;

            //等号的左边：数组中键值对的哈希值
            //等号的右边：当前要添加键值对的哈希值
            //如果键不一样，此时返回false
            //如果键一样，返回true
            boolean b1 = p.hash == hash;

            if (b1 && ((k = p.key) == key || (key != null && key.equals(k)))){
                e = p;
            } else if (p instanceof TreeNode){
                //判断数组中获取出来的键值对是不是红黑树中的节点
                //如果是，则调用方法putTreeVal，把当前的节点按照红黑树的规则添加到树当中。
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            } else {
                //如果从数组中获取出来的键值对不是红黑树中的节点
                //表示此时下面挂的是链表
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        //此时就会创建一个新的节点，挂在下面形成链表
                        p.next = newNode(hash, key, value, null);
                        //判断当前链表长度是否超过8，如果超过8，就会调用方法treeifyBin
                        //treeifyBin方法的底层还会继续判断
                        //判断数组的长度是否大于等于64
                        //如果同时满足这两个条件，就会把这个链表转成红黑树
                        if (binCount >= TREEIFY_THRESHOLD - 1)
                            treeifyBin(tab, hash);
                        break;
                    }
                    //e:            0x0044    ddd    444
                    //要添加的元素：  0x0055    ddd    555
                    //如果哈希值一样，就会调用equals方法比较内部的属性值是否相同
                    if (e.hash == hash && ((k = e.key) == key || (key != null &&
key.equals(k)))){
                         break;
                    }

                    p = e;
                }
            }

            //如果e为null，表示当前不需要覆盖任何元素
            //如果e不为null，表示当前的键是一样的，值会被覆盖
            //e:0x0044  ddd  555
```

```
            //要添加的元素：  0x0055   ddd   555
            if (e != null) {
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null){

                    //等号的右边：当前要添加的值
                    //等号的左边：0x0044的值
                    e.value = value;
                }
                afterNodeAccess(e);
                return oldValue;
            }
        }

        //threshold：记录的就是数组的长度 * 0.75，哈希表的扩容时机  16 * 0.75 = 12
        if (++size > threshold){
             resize();
        }

        //表示当前没有覆盖任何元素，返回null
        return null;
    }
```

创建不可变集合的场景

如果某一个数据不能被修改的时候，把他防御性的拷贝到我们的不可变的集合中

当集合的中的对象被不可信的库调用的时候，不可变的形式的是安全的

创建不可变集合的书写的格式

再list Set Map的接口中，都存在静态的of方法，可以获取到一个不可变的集合

```
static<E> List<E> of(E....elements)
static<E> Set<E> of(E....elements)
static<K,v> Map<K,v> of(E....elements)
```

该集合不能添加，不能删除。

```
public class demo1 {
    public static void main(String[] args) {
        List<String> hello = List.of("HELLO", "WORK");
        System.out.println(hello);
        hello.add("ddd");
        System.out.println(hello);
    }
}
```

创建成功之后就不能再删减增加了，强行的使用就会出现报错

Stream流

```
public class demo1 {
    public static void main(String[] args) {
      ArrayList<String> list=new ArrayList<>();
      list.add("张无忌");
      list.add("周芷若");
      list.add("赵敏");
      list.add("张强");
      list.add("张三丰");
//      ArrayList<String> list2=new ArrayList<>();
//        for (String s : list) {
//            if (s.startsWith("张")){
//               list2.add(s);
//            }
//        }
//        System.out.println(list2);
//
//        ArrayList<String> list3=new ArrayList<>();
//        for (String s : list2) {
//            if (s.length()==3){
//                list3.add(s);
//            }
//        }
//        System.out.println(list3);
//        for (String s : list3) {
//            System.out.println(s);
//        }

   list.stream().filter(name->name.startsWith("张")).filter(name-
>name.length()==3).forEach(name-> System.out.println(name));

    }
}
初尝试stream 流
```

stream 流:

流：流水线

结合lamada的表达式去简化集合和数组的一些的操作。

使用的步骤;

```
先得到一个stream的数据流水线，并将我们的数据放上去。
利用stream流中的API进行各种的操作
过滤    转换     统计   打印。
```

使用的步骤

```

先得到一条流水线，并把数据放到上面
```

```
使用中间方法对流水线上的数据进行操作

使用终结的方法对流水线上的数据操作


   创建数据流，将数据放到对应的数据流上
   Stream<String> stream = list.stream();
使用中间的方法处理stream 的流
   Stream<String> stringStream = stream.filter(s -> s.length() >= 3);
使用collect收集数据流
   List<String> collect = stringStream.collect(Collectors.toList());
   System.out.println(collect);
```

stream 的流的使用的步骤一：得到一条stream的流水线，并把数据放上去

```
获取的方法            方法名字          说明

单列的集合            stream()          COLLECTION中的默认的方法
双列的集合             无                 无法使用数据流
数组          Public static stream()     Arrays工具类中的静态的方法
一堆零散的数据      Public static Stream<T> of(T..values)  stream接口中的静态的方法
```

单列的集合

```
public class demo1 {
    public static void main(String[] args) {
      ArrayList<String> list=new ArrayList<>();
      list.add("张无忌");
      list.add("周芷若");
      list.add("赵敏");
      list.add("张强");
      list.add("张三丰");

      list.stream().filter(name->name.startsWith("张")).filter(name-
>name.length()==3).forEach(name-> System.out.println(name));

    }
}
```

数组:

```
        Integer[] arr=new Integer[10];
        for (int i = 0; i < arr.length; i++) {
            arr[i]=1;
        }
        Arrays.stream(arr).forEach(s-> System.out.println(s));
    }
}
```

零散的数据：

```
public class demo3 {
    public static void main(String[] args) {

        Stream.*of*(1,2,3,4,5,6,7,8).forEach(s-> System.*out*.println(s));
    }
}
```

需要注意的细节

```
public class demo3 {
    public static void main(String[] args) {

        int[] arr1={1,2,3,4,5,6,7,8,9};
        String[] arr2={"q","n","c"};
        Stream.of(1,2,3,4,5,6,7,8).forEach(s-> System.out.println(s));
        Stream.of(arr1).forEach(num-> System.out.println(num));
        Stream.of(arr2).forEach(num-> System.out.println(num));
    }
}
```

方法的形参是一个可变的参数，可以传递一堆零散的数据，也可以传递数组
但是数组必须是引用数据类型的，如过传递的是基本的数据类型的数据，会把整个的数组当成一个元素，放到stream的流中。

stream的中间方法：

注意事项：

```
流只能使用一次特别重要；；；；；
```

中间方法，返回新的stream流，原来的流只使用一次，建议使用链式编程

修改stream流中的数据，不会影响原来集合或者数组中的数据

删选与切片：

```
        filter：过滤流中的某些元素
        limit(n)：获取n个元素
        skip(n)：跳过n元素，配合limit(n)可实现分页
        distinct：通过流中元素的 hashCode() 和 equals() 去除重复元素
```

```
Stream<Integer> stream = Stream.of(6, 4, 6, 7, 3, 9, 8, 10, 12, 14, 14);

Stream<Integer> newStream = stream.filter(s -> s > 5) //6 6 7 9 8 10 12 14 14
        .distinct() //6 7 9 8 10 12 14
        .skip(2) //9 8 10 12 14
        .limit(2); //9 8
newStream.forEach(System.out::println);
```

映射

```
        map：接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。
        flatMap：接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。
```

```
List<String> list = Arrays.asList("a,b,c", "1,2,3");

//将每个元素转成一个新的且不带逗号的元素，接受的是一个函数，整个函数是将会作用域我们所有的元素
Stream<String> s1 = list.stream().map(s -> s.replaceAll(",", ""));
s1.forEach(System.out::println); // abc  123

Stream<String> s3 = list.stream().flatMap(s -> {
    //将每个元素转换成一个stream
    String[] split = s.split(",");
    Stream<String> s2 = Arrays.stream(split);
    return s2;
});
s3.forEach(System.out::println); // a b c 1 2 3
```

stream的终结：

```
void forEach(Consume action)    对此流的每个元素执行操作
long count()    返回此流中的元素数
toArray()       收集流中的数据，放到数组中
```

Stream流的收集操作

```
public static Collector toList()    把元素收集到List集合中
public static Collector toSet() 把元素收集到Set集合中
public static Collector toMap(Function keyMapper,Function valueMapper)  把元素收集到Map集合中
```

for example

```
public class demo5 {
    public static void main(String[] args) {

                //创建List集合对象

                List<String> list = new ArrayList<String>();

                list.add("孙悟空");
                list.add("孙行者");
                list.add("王宝强");
                list.add("柳神");
                //需求1：得到名字为3个字的流
                Stream<String> listStream = list.stream().filter(s -> s.length() == 3);
                //需求2：把使用Stream流操作完毕的数据收集到List集合中并遍历
                List<String> names = listStream.collect(Collectors.toList());
                for(String name : names) {
                    System.out.println(name);
                }
                //创建Set集合对象
                Set<Integer> set = new HashSet<Integer>();
                set.add(10);
                set.add(20);
                set.add(30);
                set.add(33);
                set.add(35);

                //需求3：得到年龄大于25的流
                Stream<Integer> setStream = set.stream().filter(age -> age > 25);

                //需求4：把使用Stream流操作完毕的数据收集到Set集合中并遍历
                Set<Integer> ages = setStream.collect(Collectors.toSet());
                for(Integer age : ages) {
                    System.out.println(age);
                }

                //定义一个字符串数组，每一个字符串数据由姓名数据和年龄数据组合而成
                String[] strArray = {"孙悟空,560", "孙行者,555", "王宝强,33", "柳神,22"};
```

```
                //需求5：得到字符串中年龄数据大于28的流
                Stream<String> arrayStream = Stream.of(strArray).filter(s ->
Integer.parseInt(s.split(",")[1]) > 28);

                //需求6：把使用Stream流操作完毕的数据收集到Map集合中并遍历，字符串中的姓名作键，
年龄作值
          Map<String, Integer> map = arrayStream.collect(Collectors.toMap(s ->
s.split(",")[0], s -> Integer.parseInt(s.split(",")[1])));

                Set<String> keySet = map.keySet();
                for (String key : keySet) {
                    Integer value = map.get(key);
                    System.out.println(key + "," + value);
                }
            }
        }
```

收集 的方法：

```
public class demo6 {
    public static void main(String[] args) {
        ArrayList<String> list=new ArrayList<>();
        Collections.addAll(list,"张无忌-男-15","周芷若-女-24","赵敏-女-12","张强-
男-20","张三丰-男-100","张翠山-男-40","张良-男-35","王二麻子-男-37","谢广坤-男-61");
        System.out.println(list.toString());
        System.out.println(list.stream().filter(s -> s.split("-")
[1].equals("男")).filter(s -> Integer.parseInt(s.split("-")[2]) >=
18).collect(Collectors.toList()));
        System.out.println(list.stream().filter(s -> s.split("-")
[1].equals("男")).filter(s -> Integer.parseInt(s.split("-")[2]) >=
18).collect(Collectors.toSet()));
        Map<String, String> mymap = list.stream().filter(s -> s.split("-")
[1].equals("男")).filter(s -> Integer.parseInt(s.split("-")[2]) >=
18).collect(Collectors.toMap(s -> s.split("-")[0], s -> s.split("-")[1]));
        for (Map.Entry<String, String> stringStringEntry : mymap.entrySet()) {
            System.out.println(stringStringEntry.getKey()+"
"+stringStringEntry.getValue());
        }
    }
```

方法引用：

方法引用：

用以前学习的方法，拿过来用，当作函数式接口中抽象方法的方法体

```
public class demo7 {
    public static void main(String[] args) {
        Integer[] myarr=new Integer[10];
        for (int q = 0; q < 10; q++) {
```

```
            myarr[q]=q;
        }
        Arrays.sort(myarr, new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2-o1;
            }
        });
        for (Integer integer : myarr) {
            System.out.println(integer);
        }
    }
}
```

比如这个compare

```
  Arrays.sort(myarr, new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2-o1;
            }
        });
```

出了使用函数式接口完成，还能够使用方法的引用完成也就是：

```
    new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2-o1;
            }
        }
```

这个一部分可以用方法的引用替换完成

```
    public  int subtraction(int n1,int n2){
    return n2-n1;
}
```

使用的规则：

首先引用的部分必须树函数式的接口
其次被引用的方法必须已经存在
被引用的方法的形参和返回值需要和抽象方法保持一致

例如：

首先这个引用的位置是一个函数式的接口

```
    new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2-o1;
            }
        }
```

被引用的方法已经存在

被引用的方法的形参和返回值需要和抽象的方保持一致

```
    public  int subtraction(int n1,int n2){
    return n2-n1;
}
```

举例:

```
::    独特的方法引用符
```

```
package com.itheima.test16;

import java.awt.event.ActionListener;
import java.util.*;
import java.util.stream.Collectors;

public class demo7 {
    public static void main(String[] args) {
        Integer[] myarr=new Integer[10];
        for (int q = 0; q < 10; q++) {
            myarr[q]=q;
        }
//        匿名内部类
        Arrays.sort(myarr, new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2-o1;
            }
        });
        for (Integer integer : myarr) {
            System.out.println(integer);
        }

//        lamada表达式
        Arrays.sort(myarr, (o1,o2)-> o1-o2);
        for (Integer integer : myarr) {
            System.out.println(integer);
        }
//        方法的引用
//        通过类名的引用
//        将该方法作为抽象方法的方法体

        Arrays.sort(myarr,demo7::subtraction);
        for (Integer integer : myarr) {
            System.out.println(integer);
        }
```

```
    }
    public  static int subtraction(int num1,int num2){
        return num1-num2;
    }
}
```

方法引用分类

引用静态方法

格式:

```
类名:: 静态方法名
Integer::parseInt
```

举例:

```
        Arrays.sort(myarr,demo7::subtraction);
        for (Integer integer : myarr) {
            System.out.println(integer);
        }

    }
    public  static int subtraction(int num1,int num2){
        return num1-num2;
    }
}
```

example:

使用匿名内部类完成的操作

```
public class demo1 {
    public static void main(String[] args) {

        ArrayList<String> list=new ArrayList<>();
        Collections.addAll(list,"1","2","3","4","5","6","7","8");
        list.stream().map(new Function<String, Integer>() {
            @Override
            public Integer apply(String s) {
               int i=Integer.parseInt(s);
               return i;
            }
        }).forEach(s-> System.out.println(s));
    }
}
```

使用lamada完成

```
public class demo1 {
    public static void main(String[] args) {

        ArrayList<String> list=new ArrayList<>();
        Collections.addAll(list,"1","2","3","4","5","6","7","8");
        list.stream().map(s->Integer.parseInt(s)).forEach(s->
System.out.println(s));
    }
}
```

使用我们的方法引用

```
public class demo1 {
    public static void main(String[] args) {

        ArrayList<String> list=new ArrayList<>();
        Collections.addAll(list,"1","2","3","4","5","6","7","8");
        list.stream().map(Integer::parseInt).forEach(s-> System.out.println(s));
    }
}
```

引用成员方法

格式:

```
格式：对象:: 成员方法
```

引用其他类的成员方法

```
其他类对象名::方法名
```

```
        ArrayList<String> list=new ArrayList<>();
        Collections.addAll(list,"张三","张无忌","周芷若","赵敏","张强","张三丰");
//        给成方法引用
        list.stream().filter(s->s.startsWith("张")).filter(s-
>s.length()>=3).forEach(s -> System.out.println(s));
        list.stream().filter(demo1::mytest).forEach(s -> System.out.println(s));
```

引用本类的成员方法

```
this::方法名
```

```
public class demo1 {
    public static void main(String[] args) {
```

```
        ArrayList<String> list=new ArrayList<>();
        Collections.addAll(list,"张三","张无忌","周芷若","赵敏","张强","张三丰");
//        给成方法引用
list.stream().filter(s->s.startsWith("张")).filter(s->s.length()>=3).forEach(s ->
System.out.println(s));

list.stream().filter(new demo1()::mytest).forEach(s -> System.out.println(s));
    }
public    boolean mytest(String s){
        if(s.startsWith("张")&s.length()==3){
            return true;

        }else
            return false;

    }
}
```

######

举例子:

```
        x1.addActionListener(this::actionPerformed1);
        x2.addActionListener(this::actionPerformed2);
        x3.addActionListener(this::actionPerformed3);
        this.getContentPane().add(x1);
        this.getContentPane().add(x2);
        this.getContentPane().add(x3);
//        this.addMouseListener();
        //让界面显示出来，默认页面是不显示的
        this.setVisible(true);

    }

    public void actionPerformed1(ActionEvent e) {
        System.out.println("登陆键被点击了");

    }

    public void actionPerformed2(ActionEvent e) {
        System.out.println("注册键被点击了");

    }

    public void actionPerformed3(ActionEvent e) {
        System.out.println("斗地主被点击了");

    }

}
```

用父类的成员方法

```
super::方法名
```

```
        x1.addActionListener(super::actionPerformed1);
        x2.addActionListener(super::actionPerformed2);
        x3.addActionListener(super::actionPerformed3);
        this.getContentPane().add(x1);
        this.getContentPane().add(x2);
        this.getContentPane().add(x3);
```

```
public class mufram extends JFrame {

    public void actionPerformed1(ActionEvent e) {
        System.out.println("登陆键被点击了");
    }

    public void actionPerformed2(ActionEvent e) {
        System.out.println("注册键被点击了");
    }

    public void actionPerformed3(ActionEvent e) {
        System.out.println("斗地主被点击了");
    }

    public String toString() {
        return "mufram{}";
    }
}
```

引用构造方法:

我们可以将流中的每一个数据去直接引用构造方法实现我们每一个类的带参创建

要求是将我们的带参的构造实现后完成一定的功能即也就是我们需要完成一个具体的实现，比如我们这个构造方法的接受的参数是我们流中的每一个元素，我们将这个元素的数值提取出来赋值给类中的变量，完成对象的创建即可

格式:

```
类名::new
```

范例:

```
student::new
```

类中重新定义一个构造方法:

```
    public student(String s) {
        this.name=s.split(",")[0];
        this.age=Integer.parseInt(s.split(",")[1]);
    }
```

引用类中的构造方法：

```
public class demo3 {
    public static void main(String[] args) {
        ArrayList<String> myarr1 = new ArrayList<>();
        Collections.addAll(myarr1, "张三风,23", "李四能,24", "王开五,25","赵六,32","黄豆,20","刘光,23");
//        把stream流中的的string封装成一个student
//        方法的引用
//         myarr1.stream().map(demo3::apply1).forEach(s-> System.out.println(s));

        myarr1.stream().map(student::new).forEach(s-> System.out.println(s));

    }
    public static student apply1(String s) {
        String name=s.split(",")[0];
        int s2=Integer.parseInt(s.split(",")[1]);
        return new student(name,s2);

    }
}
```

其他的调用的方式：

使用类名引用成员方法：

自己调用自己的无参的；

被引用的方法的形参是跟抽象杨方法中的第二个参数后面的保持一致

局限性：

不能引用所有的类中的 成员方法

是跟抽象方法中的第一个的参数是相关的，有什么的类型，就只能引用这个类中的成员方法

```
格式：类名::成员方法
```

```
范例：  String:: substring
```

```
public class demo4 {
    public static void main(String[] args) {
        ArrayList<String> list=new ArrayList<>();
        Collections.addAll(list,"aaa","bbb","ccc","ddd");
//         变成大写后输出
```

```
        list.stream().map(new Function<String, String>() {
            @Override
            public String apply(String s) {
                return s.toUpperCase();
            }
        }).forEach(s -> System.out.println(s));
        list.stream().map(String::toUpperCase).forEach(s -> System.out.println(s));
    }
}
```

原始的使用匿名内部类的方法：

```
public class demo5 {

    public static void main(String[] args) {

        ArrayList<Integer> arr=new ArrayList<>();
        Collections.addAll(arr,1,2,3,4,5,6,7,8,9,10);
//        储存一些整数，收集到数组当中
        Integer[] array = arr.stream().toArray(new IntFunction<Integer[]>() {
            @Override
            public Integer[] apply(int value) {
                return new Integer[value];
            }
        });
        System.out.println(Arrays.toString(array));
    }
}
```

使用我们引用数组的构造方法：

需要的注意点在于：

我们需要了解整个数组的的类型需要和我们的arraylist的集合中的元素的类型保持一致

```
public class demo5 {
    public static void main(String[] args) {

        ArrayList<Integer> arr=new ArrayList<>();
        Collections.addAll(arr,1,2,3,4,5,6,7,8,9,10);
//        储存一些整数，收集到数组当中
        Integer[] array = arr.stream().toArray(Integer[]::new);
        System.out.println(Arrays.toString(array));
    }
}
```

练习：

```
public class demo6 {
    public static void main(String[] args) {
        ArrayList<String> myarr1 = new ArrayList<>();
        Collections.addAll(myarr1, "张三风,23", "李四能,24", "王开五,25","赵六,32","黄
豆,20","刘光,23");
        myarr1.stream().map(student::new).forEach(s-> System.out.println(s));
        List<student> collect =
myarr1.stream().map(student::new).collect(Collectors.toList());
        System.out.println(collect.toString());
//将创建的学生对象的集合中获取名字存放到数组当中
        String[] array = collect.stream().map((student)-
>student.getName()).toArray(String[]::new);
        System.out.println(Arrays.toString(array));
    }
}
```

collect.stream().map(student::getName)

第一个完成的是利用student方法将其中的Getname拿出来

第二个的操作完成的是引用数组的构造方法完成转换

```
public class demo6 {
    public static void main(String[] args) {
        ArrayList<String> myarr1 = new ArrayList<>();
        Collections.addAll(myarr1, "张三风,23", "李四能,24", "王开五,25","赵六,32","黄
豆,20","刘光,23");
        myarr1.stream().map(student::new).forEach(s-> System.out.println(s));
        List<student> collect =
myarr1.stream().map(student::new).collect(Collectors.toList());
        System.out.println(collect.toString());
        String[] array =
collect.stream().map(student::getName).toArray(String[]::new);
        System.out.println(Arrays.toString(array));

    }
}
```

异常, File,综合案例:

异常：

- 指的是程序在执行过程中，出现的非正常的情况，最终会导致JVM的非正常停止。

在Java等面向对象的编程语言中，异常本身是一个类，产生异常就是创建异常对象并抛出了一个异常对象。

Java处理异常的方式是中断处理。

> 异常指的并不是语法错误,语法错了,编译不通过,不会产生字节码文件,根本不能运行.:

异常体系

异常机制其实是帮助我们**找到**程序中的问题，异常的根类是`java.lang.Throwable`，其下有两个子类：`java.lang.Error`与`java.lang.Exception`，平常所说的异常指`java.lang.Exception`。

Throwable体系：

- **Error**:严重错误Error，无法通过处理的错误，只能事先避免，好比绝症。
- **Exception**:表示异常，异常产生后程序员可以通过代码的方式纠正，使程序继续运行，是必须要处理的。好比感冒、阑尾炎。

Throwable中的常用方法：

- `public void printStackTrace()`:打印异常的详细信息。

  包含了异常的类型,异常的原因,还包括异常出现的位置,在开发和调试阶段,都得使用printStackTrace。
- `public String getMessage()`:获取发生异常的原因。

  提示给用户的时候,就提示错误原因。
- `public String toString()`:获取异常的类型和异常描述信息(不用)。

我们平常说的异常就是指Exception，因为这类异常一旦出现，我们就要对代码进行更正，修复程序。

异常(Exception)的分类:

根据在编译时期还是运行时期去检查异常?

- **编译时期异常**:checked异常。在编译时期,就会检查,如果没有处理异常,则编译失败。(如日期格式化异常)
- 关键在于检查本地的信息

```java
public class demo7 {
    public static void main(String[] args) {
        String time="2030年1月1日";
        SimpleDateFormat sdf=new SimpleDateFormat( pattern: "yyyy年MM月dd日");
        Date mydate=sdf.parse(time);
        System.out.println(mydate);
    }
}
```

此时会出现的编译时期的异常整个就意味着我们需要处理该异常，否则无法编译

- **运行时期异常**:runtime异常。在运行时期,检查异常.在编译时期,运行异常不会编译器检测(不报错)。(如数学异常)
- 关键在于检查运行的异常
-

```java
        int[] arr={1,2,3,4,5,6};
        System.out.println(arr[10]);
    }
}
```

此时的异常

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException Create breakpoint : Index 10 out of bounds for length 6
    at com.itheima.TEST17.demo7.main(demo7.java:15)
```

`ArrayIndexOutOfBoundsException` 数组越界异常

异常的作用：

作用一：可以用来查询我们的bug的关键参考的信息、

作用二：异常可以作为方法内部的一种特殊的返回值，以便通知调用者底层的执行的情况；

抛出异常throw：

在编写程序时，我们必须要考虑程序出现问题的情况。比如，在定义方法时，方法需要接受参数。那么，当调用方法使用接受到的参数时，首先需要先对参数数据进行合法的判断，数据若不合法，就应该告诉调用者，传递合法的数据进来。这时需要使用抛出异常的方式来告诉调用者。

在java中，提供了一个**throw**关键字，它用来抛出一个指定的异常对象。那么，抛出一个异常具体如何操作呢?

1. 创建一个异常对象。封装一些提示信息(信息可以自己编写)。
2. 需要将这个异常对象告知给调用者。怎么告知呢？怎么将这个异常对象传递到调用者处呢？通过关键字throw就可以完成。throw 异常对象。

   throw**用在方法内**，用来抛出一个异常对象，将这个异常对象传递到调用者处，并结束当前方法的执行。

使用的格式：

```
throw new 异常类名(参数);
```

例如：

```
throw new NullPointerException("要访问的arr数组不存在");
throw new ArrayIndexOutOfBoundsException("该索引在数组中不存在，已超出范围");
```

声明异常throws

声明异常：将问题标识出来，报告给调用者。如果方法内通过throw抛出了编译时异常，而没有捕获处理（稍后讲解该方式），那么必须通过throws进行声明，让调用者去处理。

关键字**throws**运用于方法声明之上,用于表示当前方法不处理异常,而是提醒该方法的调用者来处理异常(抛出异常).

```
public class demo8 {
    public static void main(String[] args) {
        int[] arr = new int[10];
        Arrays.setAll(arr, operand -> 0);
        Random rd = new Random();
        for (int i = 0; i < arr.length; i++) {
            arr[i] = rd.nextInt(arr.length);
        }
        int[] arr1=new int[0];
        try {
            int max=getmax(arr1);
            System.out.println(max);
        } catch (NullPointerException e) {
           e.printStackTrace();
        } catch (ArrayIndexOutOfBoundsException e){
           e.printStackTrace();
        }


    }
    private static int getmax(int[] arr) {
        if (arr==null){
            throw new  NullPointerException("要访问的arr数组不存在");
        }
        if (arr.length==0){
            throw new ArrayIndexOutOfBoundsException();
        }

        System.out.println("看看我执行了吗");
        int asInt = Arrays.stream(arr).max().getAsInt();
        return asInt;
    }
}
```

捕获异常

try...catch

如果异常出现的话,会立刻终止程序,所以我们得处理异常:

1. 该方法不处理,而是声明抛出,由该方法的调用者来处理(throws)。
2. 在方法中使用try-catch的语句块来处理异常。

try-catch的方式就是捕获异常。

- **捕获异常**：Java中对异常有针对性的语句进行捕获，可以对出现的异常进行指定方式的处理。

捕获异常语法如下：

```
try{
      编写可能会出现异常的代码
}catch(异常类型  e){
      处理异常的代码
      //记录日志/打印异常信息/继续抛出异常
}
```

try：该代码块中编写可能产生异常的代码。

catch：用来进行某种异常的捕获，实现对捕获到的异常进行处理。

> 注意:try和catch都不能单独使用,必须连用。

演示如下：

```
public class TryCatchDemo {
    public static void main(String[] args) {
        try {// 当产生异常时，必须有处理方式。要么捕获，要么声明。
            read("b.txt");
        } catch (FileNotFoundException e) {// 括号中需要定义什么呢？
            //try中抛出的是什么异常，在括号中就定义什么异常类型
            System.out.println(e);
        }
        System.out.println("over");
    }
    /*
     *
     * 我们 当前的这个方法中 有异常  有编译期异常
     */
    public static void read(String path) throws FileNotFoundException {
        if (!path.equals("a.txt")) {//如果不是 a.txt这个文件
            // 我假设  如果不是 a.txt 认为 该文件不存在 是一个错误 也就是异常  throw
            throw new FileNotFoundException("文件不存在");
        }
    }
```

```
}
```

如何获取异常信息：

Throwable类中定义了一些查看方法:

- `public String getMessage()`:获取异常的描述信息,原因(提示给用户的时候,就提示错误原因。
- `public String toString()`:获取异常的类型和异常描述信息(不用)。
- `public void printStackTrace()`:打印异常的跟踪栈信息并输出到控制台。

```
int[] arr={1,2,3,4,5,6};
        try {
            System.out.println(arr[10]);
        }
        catch (Exception e){
            e.printStackTrace();
        }
        System.out.println("看我检测完了吗");
```

报告的异常：

```
java.lang.ArrayIndexOutOfBoundsException Create breakpoint : Index 10 out of bounds for length 6
    at com.itheima.TEST17.demo7.main(demo7.java:16)
```

包含了异常的类型,异常的原因,还包括异常出现的位置,在开发和调试阶段,都得使用printStackTrace。

在开发中呢也可以在catch将编译期异常转换成运行期异常处理。

多个异常使用捕获又该如何处理呢?

1. 多个异常分别处理。
2. 多个异常一次捕获，多次处理。
3. 多个异常一次捕获一次处理。

一般我们是使用一次捕获多次处理方式，格式如下：

```
try{
      编写可能会出现异常的代码
}catch(异常类型A  e){  当try中出现A类型异常,就用该catch来捕获.
      处理异常的代码
      //记录日志/打印异常信息/继续抛出异常
}catch(异常类型B  e){  当try中出现B类型异常,就用该catch来捕获.
      处理异常的代码
      //记录日志/打印异常信息/继续抛出异常
}
```

> 注意:这种异常处理方式，要求多个catch中的异常不能相同，并且若catch中的多个异常之间有子父类异常的关系，那么子类异常要求在上面的catch处理，父类异常在下面的catch处理。

自定义的异常：

自定义的异常的继承

```
继承：  RuntimeException
核心表示由于参数的问题导致的错误

继承：  Exception
核心  提醒程序员检查本地的信息
```

首先定义一个异常的登陆类

```
// 业务逻辑异常
public class LoginException extends Exception {
    /**
     * 空参构造
     */
    public LoginException() {
    }

    /**
     *
     * @param message 表示异常提示
     */
    public LoginException(String message) {
        super(message);
    }

}
```

测试类：

```
package com.itheima.test18;

public class Demo {
    // 模拟数据库中已存在账号
    private static String[] names = {"bill","hill","jill"};

    public static void main(String[] args) {
        //调用方法
        try{
            // 可能出现异常的代码
            checkUsername("bill");
            System.out.println("注册成功");//如果没有异常就是注册成功
        } catch(LoginException e) {
            //处理异常
            e.printStackTrace();
        }
    }
```

```
    //判断当前注册账号是否存在
    //因为是编译期异常，又想调用者去处理 所以声明该异常
    public static boolean checkUsername(String uname) throws LoginException {
        for (String name : names) {
            if(name.equals(uname)){//如果名字在这里面 就抛出登陆异常
                throw new LoginException("亲"+name+"已经被注册了！");
            }
        }
        return true;
    }
}
```

File的学习

路径:

相对路径

绝对路径

File

> 文件和目录路径名的抽象表示形式。

构造方法

```
File(File parent, String child)
          根据 parent File的父路径名和 child 路径名字符串创建一个新 File 实例。
File(String pathname)
          通过将给定路径名字符串转换为抽象路径名来创建一个新 File 实例。
File(String parent, String child)
          根据 parent 路径名字符串和 child 路径名字符串创建一个新 File 实例。
File(URI uri)
          通过将给定的 file: URI 转换为一个抽象路径名来创建一个新的 File 实例。
```

第一个字符串的路径对象的创建:

```
import java.io.File;

public class File_demo1 {

    public static void main(String[] args) {
        String str="C:\\Users\\Administrator\\IdeaProjects\\mystudy\\src";
        File f1=new File(str);
        System.out.println(f1);
    }
}
```
输出:
```
    C:\Users\Administrator\IdeaProjects\mystudy\src
```

根据父字符串的和子字符串的地址创建的File 的对象

```
public class File_demo2 {

    public static void main(String[] args) {
        String str="C:\\Users\\Administrator\\IdeaProjects\\mystudy\\src";
        String str1="com\\itheima";
        File f2=new File(str,str1);
        System.out.println(f2);

    }
}
```

根据File的对象和子字符串构建File的对象

```
public class File_demo2 {
    public static void main(String[] args) {
        String str="C:\\Users\\Administrator\\IdeaProjects\\mystudy\\src";
        String str1="com\\itheima";
        File f1=new File(str);
        File f2=new File(f1,str1);
        System.out.println(f2);

    }
}
```

常见的成员的方法

判断和获取相关的方法

```
方法  描述
public String getAbsolutePath() 获取绝对路径
public String getPath() 获取路径
public String getName() 获取名称
public String getParent()   获取上层文件目录路径。若无，返回null
public long length()    获取文件长度（即：字节数）。不能获取目录的长度。
public long lastModified()  获取最后一次的修改时间，毫秒值
public String[] list()  获取指定目录下的所有文件或者文件目录的字符串数组
public File[] listFiles()   获取指定目录下的所有文件或者文件目录的File对象数组
```

```
public class File_demo2 {
    public static void main(String[] args) {
        String str="C:\\Users\\Administrator\\IdeaProjects\\mystudy\\src";
        String str1="com\\itheima";
        File f1=new File(str);
        File f2=new File(f1,str1);
        System.out.println(f2);
        System.out.println(f2.getAbsoluteFile());
        System.out.println(f2.getPath());    //获取地址
        System.out.println(f2.getName());    //获取名称
        System.out.println(f2.getParent()); //获取上层的地址
        System.out.println(f2.length());  //获取文件的长度
        System.out.println(f2.lastModified());//获取最后一次的修改时间
        String[] list = f2.list();  //获取目录下的所有的文件名
        for (int i = 0; i < list.length; i++) {
            System.out.println(list[i]);
        }
        File[] files = f2.listFiles();  //获取目录下所有的路径的File对象的数组
        for (int i = 0; i < files.length; i++) {
            System.out.println(files[i]);
        }
    }
}
```

判断的方法：

```
方法  描述
public boolean isDirectory()    判断是否是文件目录
public boolean isFile() 判断是否是文件
public boolean exists() 判断是否存在
public boolean canRead()    判断是否可读
public boolean canWrite()   判断是否可写
public boolean isHidden()   判断是否隐藏
```

```
public class File_demo3 {
    public static void main(String[] args) {
        String str="C:\\Users\\Administrator\\IdeaProjects\\mystudy\\src";
        String str1="com\\itheima";
```

```
        File f1=new File(str);
        File f2=new File(f1,str1);
        System.out.println(f2);
        System.out.println(f2.isFile());//判断是否是文件
        System.out.println(f2.isDirectory());//判断是否是文件的目录
        System.out.println(f2.exists());    //判断是否存在
        System.out.println(f2.canRead());//是否可读
        System.out.println(f2.canWrite());//是否可写
        System.out.println(f2.isHidden());//是否隐藏
    }
}
```

创建功能

```
public boolean createNewFile()  创建文件。若文件存在，则不创建，返回false
public boolean mkdir()  创建文件目录。如果此文件目录存在，就不创建了。如果此文件目录的上层目录不存在，也不创建。
public boolean mkdirs() 创建文件目录。如果上层文件目录不存在，一并创建
```

```
public class File_demo5 {
    public static void main(String[] args) throws IOException {
        String str="C:\\Users\\Administrator\\IdeaProjects\\mystudy\\src";
        String str1="com\\itheima\\test_demo";
        String str2="com\\itheima\\test_demo\\my.txt";
        File f1=new File(str);
        File f2=new File(f1,str1);
        System.out.println(f2);
        if (f2.mkdir()) {
            System.out.println("创建成功");
        }else System.out.println("创建失败");
        File f3=new File(f1,str2);
        System.out.println(f3.createNewFile());
    }
}
```

注意的事项

就是我们再创建的的时候使用mkdir和mkdirs，俩种方法完成该操作的时候，我们需要注意一下：madir是不存在父级的文件夹的时候，则不创建该文件夹，而要是mkdirs是直接创建其父文件夹及其本身

综合练习：

遍历所有的文件夹下面的文件：

```
public class File_demo6 {
    public static void main(String[] args) throws IOException {
        String str = "C:\\Users\\Administrator\\IdeaProjects\\mystudy\\src";
        String str1 = "com\\itheima\\";
        File f1 = new File(str);
```

```
        File f2 = new File(f1, str1);
        System.out.println(f2);
        File[] files = f2.listFiles();
        for (File file : files) {
            System.out.println(file);
        }
        showdir(f2,0);

    }
    public static void showdir(File file, int count) {
        System.out.println("|" + file.getName());
        File[] files = file.listFiles();
        count++;
        for (File file1 : files) {
            if (file1.isFile()) {
                System.out.println(file1);
            } else
                showdir(file1, count);
        }
    }
}
```

删除的方法：

```
public boolean delete() 删除文件或者文件夹
```

注意事项： Java中的删除不走回收站。 要删除一个文件目录，请注意该文件目录内不能包含文件或者文件目录。

综合练习：查找文件下是否存在某一个文件：

```
public class File_demo7 {
    public static void main(String[] args) throws IOException {
        String str = "C:\\Users\\Administrator\\IdeaProjects\\mystudy\\src";
        String str1 = "com\\itheima\\";
        File f1 = new File(str);
        File f2 = new File(f1, str1);
        System.out.println(f2);
        File[] files = f2.listFiles();
        showdir(new File("D:\\"),0);
    }
    public static void showdir(File file, int count) {
        System.out.println("|"+file);
        File[] files = file.listFiles();
        count++;
        for (File file1 : files) {
            if (file1.isFile()) {
                if (file1.toString().split("\\.")[1].equals(new String("avi"))) {
                    System.out.println(file1.toString().split("\\.")[1]);
                    System.out.println(file1);
                }
```

```
            } else
                showdir(file1, count);
        }
    }
}
```

```
package com.itheima.test19;

import java.io.File;
import java.io.IOException;

public class File_demo7 {
    public static void main(String[] args) throws IOException {
        String str = "C:\\Users\\Administrator\\IdeaProjects\\mystudy\\src";
        String str1 = "com\\itheima\\";
        File f1 = new File(str);
        File f2 = new File(f1, str1);
        System.out.println(f2);
        File[] files = f2.listFiles();
        showdir(new File("D:\\"),0);
    }
    public static void showdir(File file, int count) {
        System.out.println("|"+file);
        File[] files = file.listFiles();
        count++;
        for (File file1 : files) {
            if (file1.isFile()) {
                if ((file1.toString().endsWith(".avi"))) {
                    System.out.println(file1);
                }
            } else
                showdir(file1, count);
        }
    }
}
```

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
//

package com.itheima.test19;

import java.io.File;
import java.io.IOException;

public class File_demo7 {
    public File_demo7() {
```

```
    }

    public static void main(String[] args) throws IOException {
        String str = "C:\\Users\\Administrator\\IdeaProjects\\mystudy\\src";
        String str1 = "com\\itheima\\";
        File f1 = new File(str);
        File f2 = new File(f1, str1);
        if (f2.mkdirs()) {
            System.out.println(f2);
        }

        deletedir(f2, 0);
    }

    public static void deletedir(File file, int count) {
        System.out.println("|" + file);
        File[] files = file.listFiles();
        ++count;
        File[] var3 = files;
        int var4 = files.length;

        for(int var5 = 0; var5 < var4; ++var5) {
            File file1 = var3[var5];
            if (file1.isDirectory()) {
                deletedir(file1, 0);
            } else {
                file1.delete();
            }
        }

    }
}
```

IO流